Facultad de informática

Universidad Complutense de Madrid

Proyecto de Sistemas Informáticos 2011-2012

Desarrollo de Agentes Adaptables para Videojuegos en Primera Persona

Autores:

José Manuel Lourido Piso

Sergio Ruiz Conde

Javier Velásquez López

Director del proyecto:

Pedro Antonio González Calero

Índice

Resumen

Este proyecto tiene como objetivo la investigación y utilización de diversas técnicas de inteligencia artificial para el desarrollo de agentes inteligentes en videojuegos en primera persona. En concreto, nos hemos centrado en dos técnicas muy conocidas, como son los árboles de comportamiento y el razonamiento basado en casos o CBR, en sus siglas en inglés.

El videojuego elegido para este proyecto ha sido Unreal Tournament 2004, el cual nos proporciona una librería muy interesante y potente para el desarrollo de agentes inteligentes. Esta librería llamada POGAMUT, nos facilita la creación e inicialización del agente en el entorno del juego a través de una API implementada en el lenguaje de programación Java. Esto nos permite centrarnos en el diseño e implementación de la inteligencia del agente.

Para llevar a cabo esta investigación, se ha hecho uso de librerías externas para cada una de las técnicas antes mencionadas.

Por una parte, la librería JBT (Java Behavior Trees) orientada al desarrollo de los árboles de comportamiento. Esta librería nos facilita el diseño y desarrollo del árbol de comportamiento, sin preocuparnos de la implementación interna del árbol.

Y por otra parte, la librería FreeCBR para la aplicación de la técnica de razonamiento basado en casos, la cual, nos ha permitido centrarnos en el diseño de los casos que definen el comportamiento de nuestro agente, sin tener que desarrollar el motor CBR.

Con todas estas herramientas, hemos desarrollado dos agentes inteligentes, uno por cada técnica utilizada, tratando de darle la mejor funcionalidad al agente inteligente, y hemos comparado ambas técnicas haciendo hincapié en las ventajas y desventajas de cada una con respecto a la otra bajo nuestro punto de vista.

Abstract

The aim of this project is the investigation and utilization of different techniques of artificial intelligence for the development of intelligent agents in first-person video games. Concretely, we have focused in two very-known techniques, as they are behaviour trees, and case-base reasoning, or CBR.

The chosen video game for this project has been Unreal Tournament 2004, which give us a very interesting and powerful library for the development of intelligent agents. This library, called Pogamut, supplies us the creation and initialization of the agent in the game's environment, through an API implemented in the programming language Java. This brings us the possibility of focusing in the design and implementation of the agent's intelligence.

To perform this research, we have used external libraries for each techniques we have just mentioned before.

On the one hand, Java Behavior Trees library, JBT, oriented to Behavior Trees development, which supplies the design and development of these structures, not caring about low level implementation of them.

On the other hand, FreeCBR library for applying the CBR technique, allowing us to focus in the design of the cases that defines the behaviour of our agent, without developing and implementing a CBR engine from scratch.

With these libraries, we have developed two intelligent agents, one for each one of the techniques mentioned above, and made a comparison between them exposing their advantages and disadvantages under our own point of view.

Palabras clave

- Unreal Tournament
- Pogamut
- CBR
- Árboles de comportamiento
- Inteligencia Artificial
- Agentes Inteligentes

¿Qué es Unreal Tournament 2004?

Unreal Tournament 2004, también conocido como *UT2004* o *UT2K4*, es un videojuego de acción en primera persona(shooter), el tercero de la saga Unreal(tras Unreal Tournament y Unreal Tournament 2003), desarrollado por Epic Games, y distribuido por Atari. Dicho juego se caracterizó en su época por unos controles nítidos, buenos gráficos, una sólida red multijugador (lo que le convirtió en un juego para uso online principalmente), y una acción trepidante.

La historia de Unreal Tournament 2004 es bastante básica: formarás parte de un torneo que enfrenta equipos de soldados de distintas razas y corporaciones que deben combatir para obtener la victoria final, bien cumpliendo una serie de objetivos, dependiendo del modo de juego, bien aniquilando a todos los demás rivales.

Con una gran variedad de mapas y modos de juego, tanto las habituales partidas DeathMatch (todos contra todos) como los modos basados en batalla por equipos, en las cuales empezaremos desde el básico combate por equipos, hasta que, avanzadas varias rondas en el torneo, llegaremos a los modos avanzados: captura de bandera, asalto, carrera de bombardeo, etc. hasta contabilizar 10 modos de juego distintos. Contaremos también con una gran cantidad de armas, desde rifles y pistolas, hasta lanzagranadas. También contaremos con la novedad de poder usar durante las batallas vehículos, una de las principales novedades con respecto a sus predecesores

Una interesante característica introducida en UT2004, que lo diferenciaba de sus predecesores en la saga, es la importante mejora de su modo offline, ya que el comportamiento y la jugabilidad de los agentes inteligentes (jugadores controlados por el ordenador que imitan el comportamiento de los jugadores humanos) ha sido mejorada ostensiblemente, convirtiéndolos en los mejores dentro del mercado de los videojuegos de acción en esa época

En el apartado gráfico podríamos destacar un excelente modelado de los personajes y las armas, en los cuales se pueden observar una gran cantidad de detalles. También el excelente uso que se le da a los efectos de luces, humo y agua, los cuales interactúan con el jugador y el escenario de forma sobresaliente.

En definitiva Unreal Tournament 2004, es un juego, que dejó huella en los amantes de los shooters, realmente novedoso entre los videojuegos de la época, y cuanto menos adictivo.

POGAMUT

Es una plataforma en Java de código abierto usada para el rápido desarrollo de comportamientos en agentes virtuales incrustados en un entorno 3D, que permite el control en múltiples entornos proporcionados por los motores de juego. Actualmente son soportados los siguientes: *Unreal Tournament 2004 (UT2004)*, *UnrealEngine2RuntimeDemo (UE2)*, *Unreal Development Kit (UDK)* y el juego *DEFCON*.

El principal objetivo era simplificar la parte de la creación del agente. La mayoría de las acciones en el medio ambiente (incluso las más complicadas, como la búsqueda de caminos y la recolección de información en la memoria del agente) se puede realizar por uno o dos comandos. Esto permite al usuario concentrar sus esfuerzos en las partes más interesantes.

Pogamut proporciona una API de Java para crear y controlar los agentes virtuales e interfaz gráfica de usuario (plug-in de NetBeans) que simplifica la depuración de los agentes.

Interfaz de Programación (API)

- Las bibliotecas Java de Pogamut proporcionan a los programadores un conjunto de objetos y métodos que permiten consultar el estado del agente en el escenario y dar órdenes.
- Es posible conectar bibliotecas Java externas que proporcionan los sistemas de toma de decisiones de los agentes, pero el puente real entre el API de Pogamut y la biblioteca externa tiene que ser programado por el usuario.
- GaviaLib es una librería dentro de Pogamut que se encarga de traducir UnrealScript y pasarlo a Java. Permite conectar agentes a casi cualquier entorno virtual.
- *UnrealScript* es un lenguaje de scripting que utilizan todos los juegos basados en UnrealEngine (*UT2004, UE2 y UDK*), se utilizó para desarrollar la conexión con GameBots.
-

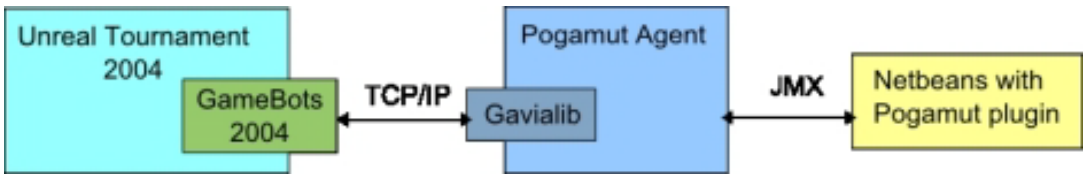


Figura 1

- La comunicación entre GameBots y Pogamut se maneja a través de un protocolo de texto a través de TCP/IP.

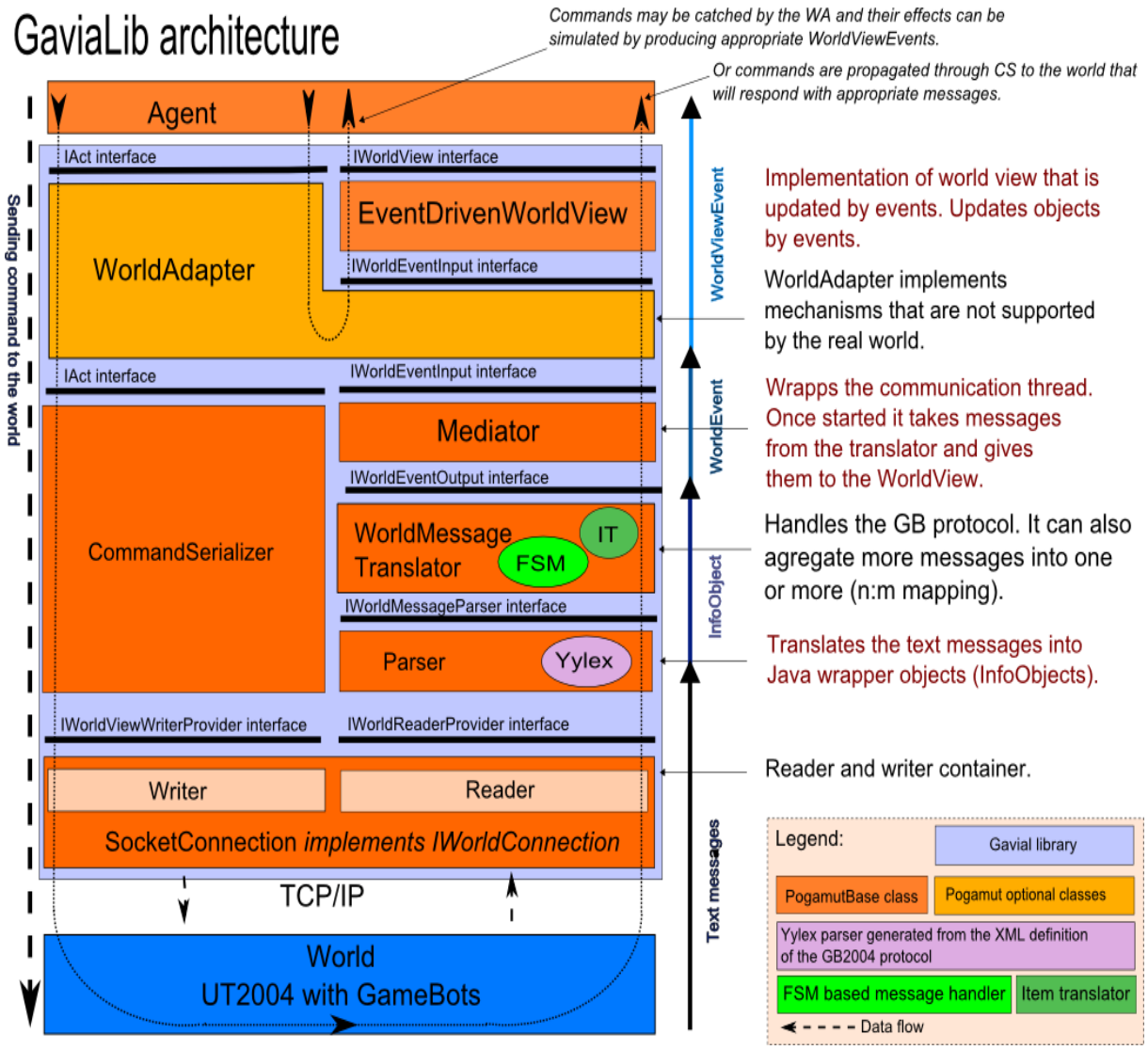


Figura 2

Instalación y configuración del sistema

A continuación se exponen los útiles necesarios para el conjunto del sistema y sus respectivos procesos de instalación. Se recomienda seguir los pasos tal y como son expuestos para evitar errores en el software.

Instalación de Unreal Tournament 2004

La instalación del producto debe hacerse en inglés para soportar la compatibilidad con *Pogamut* en la ubicación por defecto y recomendada situada en *C:\UT2004*. Tras esto, la versión de la aplicación puede no coincidir con la versión requerida para el correcto funcionamiento del sistema (v3369), la cual será actualizada posteriormente durante la configuración de *Pogamut*[1].

Instalación de NetBeans

Dado que los creadores de la plataforma *Pogamut* recomiendan el uso del *IDE NetBeans* para Java[2], es necesario instalarlo siguiendo los pasos que se van indicando en pantalla. La versión requerida es 6.9.1 JSE, disponible en la página web oficial (http://netbeans.org/downloads/6.9.1/index.html).

Instalación de Pogamut

Una vez puesto en marcha el entorno *NetBeans*, se ha de instalar la versión 3.0.11 de la herramienta Pogamut, disponible en la página web oficial (http://pogamut.cuni.cz/main/tiki-index.php?page=Old+Pogamut+3+releases).

[1] Los usuarios de Windows Vista o superior, deben descargar el software por separado para actualizar el sistema, disponible en http://www.gamershell.com/download_11747.shtml, debido a que son necesarios privilegios de administrador para realizarlo.

[2] Previamente, se ha de comprobar que se encuentra instalado Java Development Kit 1.6 o superior dado que, durante la instalación, será requerido el directorio contenedor del software.

Empty bot, cómo empezar

Introducción

Tras la instalación de todos los componentes necesarios de cara a la realización de nuestro proyecto, daríamos paso a la fase de desarrollo de los bots.

La mejor manera de empezar sería crear un nuevo proyecto, utilizando uno de los ejemplos que nos vienen dados cuando instalamos Pogamut, en este caso el empty bot. Un *empty bot*, es lo mínimo e imprescindible para crear un bot en la partida, el cual no tiene definida ninguna acción ni movimiento, siendo lo mismo que un programa "Hola Mundo".

Para crear el proyecto, entramos en NetBeans, File->New Proyect, y en esa pantalla elegimos Samples/Pogamut2004->00-EmptyBot.

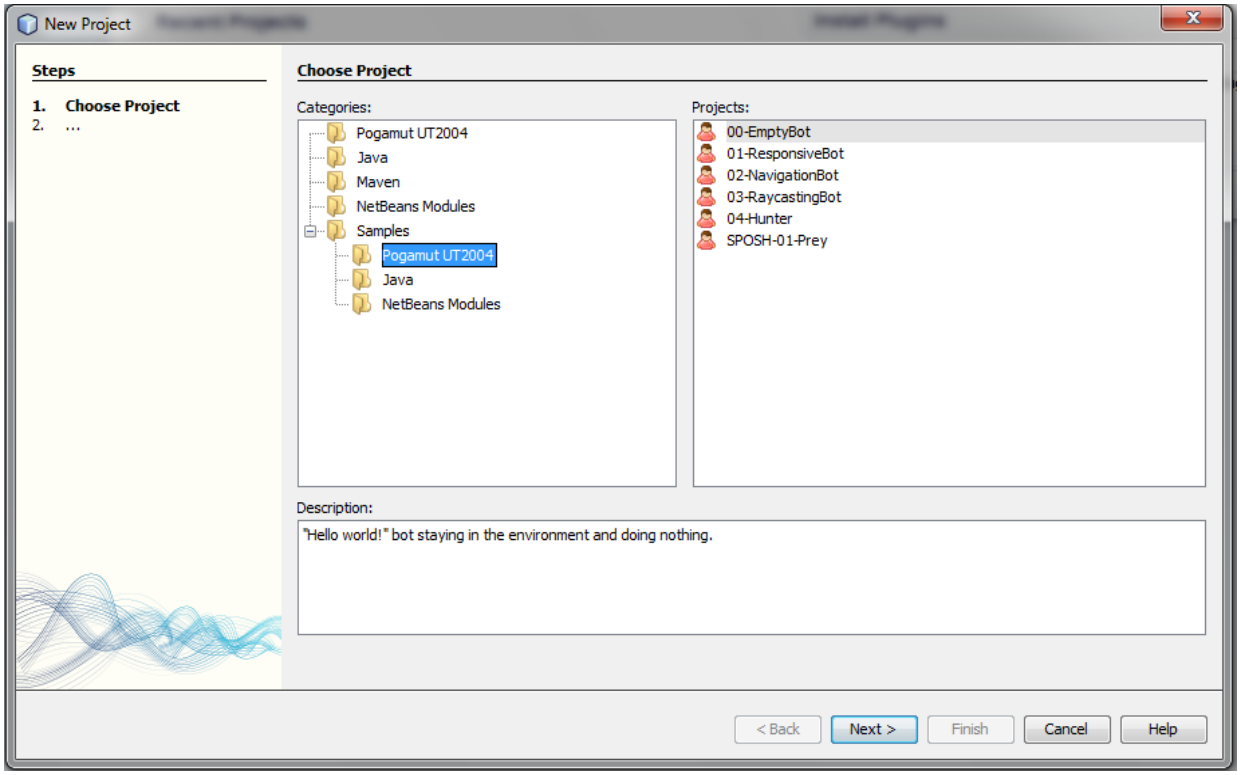


Figura 3

Ahora, si observamos la clase principal podremos ver una serie de métodos vacíos, que son los que determinarán las acciones que realizará el bot. El orden en el que se ejecutarán será:

- prepareBot(UT2004Bot bot): Método que inicializa las variables del bot, antes de que este empiece a interactuar con la partida.
- getInitializeCommand(): Método utilizado para configurar las propiedades iniciales del bot, tales como su nombre, el personaje, localización inicial, etc.
- botInitialized(GameInfo info, ConfigChange config, InitedMessage init)->
- botSpawned(GameInfo gameInfo, ConfigChange config, InitedMessage init, Self self): Método que sirve para inicializar el bot en la partida.Aquí se crea la representación física del bot en la partida. En el empty bot, también lo utilizamos para poner un mensaje de bienvenida en la partida, a imagen y semejanza del programa “Hola Mundo”.
- logic(): Método principal del bot, el cual controla cuál es la siguiente acción a realizar por él. Es llamado por el propio motor de Pogamut, es decir, es un método interno que se llama iterativamente, el cual responderá de una forma u otra dependiendo el estado de la partida.
- botKilled(BotKilled event): Método al que se accede cada vez que el bot muere. Aquí se pueden resetear todas las variables, que controlan el estado del bot.

Ahora pasaremos a ver como se vería el bot en la partida:

Antes de iniciar el bot en la partida, habría que inicializar la propia partida, por tanto comenzaremos ejecutando el servidor dedicado de UT, startGameBotsDMServer.bat , y esperar hasta que aparezca START MATCH en la consola.

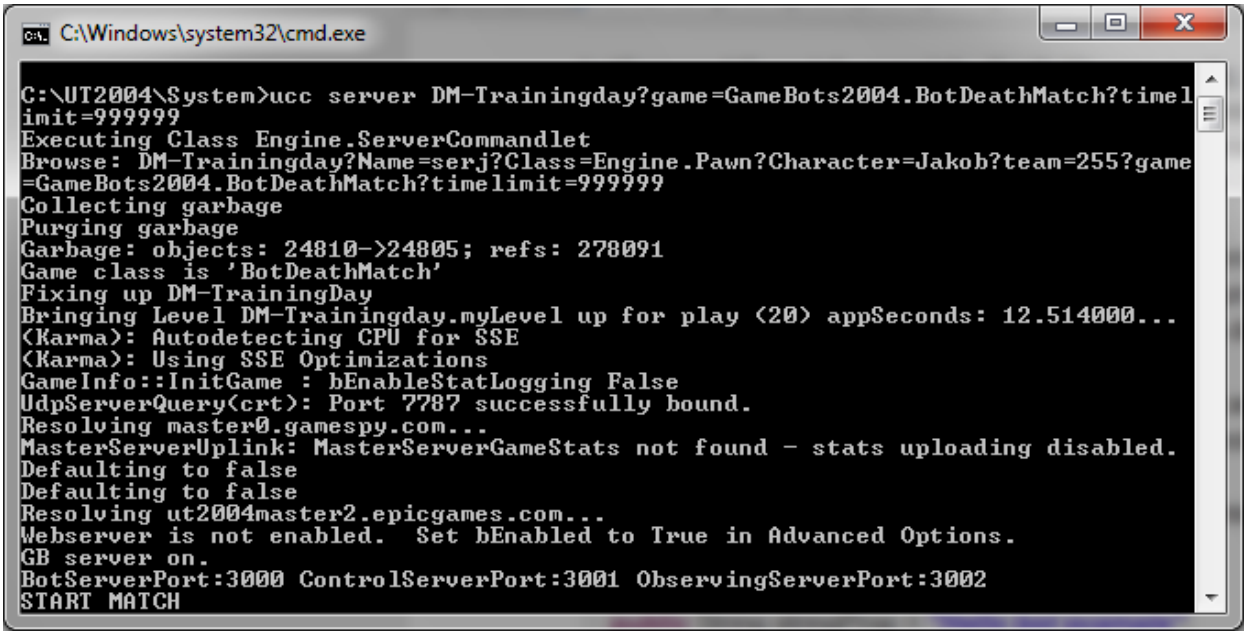


Figura 4

Despues de esto, volvermos a NetBeans, accedemos a la pestaña Servers, pulsamos el boton derecho en UT2004Servers, Add Server, y lo configuramos como indicamos en la imagen siguiente:

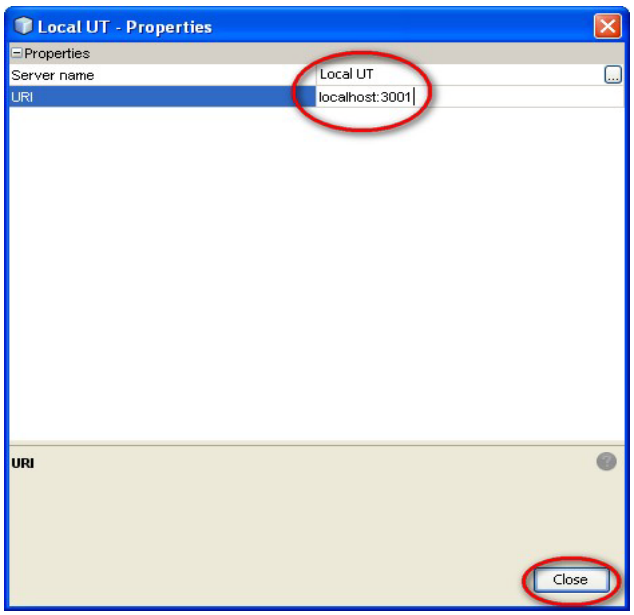


Figura 5

Tras esto si pulsamos el botón derecho sobre el servidor creado y elegimos la opción spectate, entraremos en la partida.

Para introducir un empty bot en la partida bastará con acceder a la clase principal del bot creado, y ejecutar Run File sobre ella, tras lo cual podremos ver el mensaje de bienvenida indicado en el método BotSpawned, y al propio bot dentro de la partida.

Figura 6

Movimiento, sentidos y acciones de un bot

Para que un bot pueda reaccionar o actuar a ciertas circunstancias (es atacado, se choca contra algo o alguien, etc.) debe poder sentir y moverse de alguna manera por el mundo que lo rodea. Así mismo para tomar decisiones sobre diversas acciones que se pueden llevar a cabo, como huir si se tiene poca vida o atacar si se tiene munición suficiente, necesitamos recabar información.

Obtener información

La información es siempre muy valiosa, en gestiones empresariales o en la bolsa por ejemplo, normalmente gana o prospera quien tenga la información más completa y sepa utilizarla a tiempo.

En este ámbito no es menos, y para ello primero hay que saber de qué información se dispone y cómo consultarla.

La clase *UT2004BotModuleController* [4] provee de varios componentes, que permiten consultar y enviar la información necesaria sobre el bot y el entorno, los cuales solventan muchas situaciones complicadas en las que nos hace falta conocer algún dato concreto.

Este controlador es el más avanzado que se encuentra disponible, y algunos de los módulos más complejos que podemos encontrar son los siguientes:

- Game
- AgentInfo
- Players
- Items

Game

Es un módulo especializado en la información general del juego.

Está diseñado para inicializarse dentro de la llamada al método *prepareBot()*, y para utilizarse a partir de la llamada *botInitialized()*. Esto es, porque a partir de este método ya se han recibido los mensajes suficientes desde el servidor sobre el *Mundo*.

Alguna de la información que se puede obtener a partir de este método es la siguiente:

- Tipo de juego (DeathMatch, Capturar bandera, etc.) - *getGameType().*
- Nombre del mapa - *getMapName().*
- Tiempo actual del juego, desde que empezó - *getTime().*
- Tiempo que falta para terminar - *getRemainingTime().*
- Salud con la que empezó - *getStartHealth().*

AgentInfo

Es un módulo especializado en la información general sobre el paradero del bot.

Está diseñado para inicializarse dentro de la llamada al método *prepareBot()*, y para utilizarse a partir de la llamada *botSpawned()*, ya que es cuando el bot existe en el mundo por primera vez.

La información que se puede obtener a partir de este método es toda aquella referida al bot, como puede ser:

- Su nombre - *getName().*
- Equipo - getTeam().
- Localización - *getLocation().*
- Salud - getHealth().
- Muertes que lleva - *getDeaths().*
- Situación del terreno, si esta dentro del agua - *isCurrentVolumeWater().*

Players

Es un módulo especializado en informar sobre otros jugadores. Todos los objetos de este tipo se auto-actualizan a lo largo del tiempo hasta que son destruidos.

Se puede usar a partir de *botInitialized()*, ya que es la primera vez que el bot tiene información sobre el mundo.

Este módulo permite al bot conocer todo esto sobre el resto de jugadores:

- Comprobar si puede alcanzar a otros enemigos, amigos o jugadores.
- Mapa con todos los enemigos, amigos o jugadores que son alcanzables.
- Comprobar si puede ver a otros enemigos, amigos o jugadores.
- Mapa con todos los enemigos, amigos o jugadores que son visibles.

Items

Es un módulo de memoria especializado en objetos que se encuentran en el mapa.

Se puede usar a partir de *botInitialized()*, ya que es la primera vez que el bot tiene información sobre el mundo.

Con este módulo se pueden obtener:

- Mapas con todos los ítems del escenario.

- Obtener un único ítem.
- Mapas con la lista de puntos donde salen los items.
- Mapas con la lista de puntos donde ahora mismo hay ítems disponibles.

Un pequeño ejemplo de cómo se utilizan estos métodos:

```
logic {
    Self self = getWorldView().getSingle(Self.class);
    vidaActual = self.getHealth();
    si tengoMenosVidaQueAntes(vidaActual) entonces {
        huir();
    }
}
```

En este ejemplo estamos consultando la vida que nos queda con el método *getHealth*, si esa vida disminuye pues ejecutamos la acción de "*huir*".

Eventos

Ya se dispone de la información necesaria sobre el mapa, los jugadores, el bot, los objetos, etc. y también el método de acceder a ella.

Ahora con los eventos podremos utilizarla en los momentos adecuados. Por ejemplo si el bot se encuentra de frente contra un jugador, gracias al evento *bumped* podemos averiguar si se trata de un enemigo para abatir o de un compañero.

Este tipo de comportamiento es controlado completamente por eventos, es decir, no se utiliza el método *logic()*.

Existen dos tipos de eventos, los pertenecientes a objetos y eventos que no están asociados a ningún objeto.

Pertenecientes a objetos:

- WorldObjectAppearedEvent
- WorldObjectDestroyedEvent
- WorldObjectDisappearedEvent
- WorldObjectFirstEncounteredEvent

Sin asociar a ningún objeto:

- Bumped

- MapChange
- MyInventoryEnd
- PlayerKilled

No es necesario manipular los eventos manualmente, o saber cómo iterar para seleccionar el adecuado en cada momento. Para ello, un sistema de anotaciones (*AnnotationListenerRegistrator*) permite registrar los eventos de forma muy sencilla y práctica, y el mismo se encargara de iterar entre todos los eventos registrados.

Una vez registrado el evento, se define una función que realice las acciones deseadas en cada caso.

Vamos a ver cómo utilizar un evento cuando el bot se choca contra algo o alguien.

```
@EventListener(eventClass = Bumped.class)
choca(Bumped event) {
    si chocaContraEnemigo(event) entonces {
        disparar()
    }
}
```

El sistema de anotación "*@EventListener*" nos permite indicar el tipo de evento que se espera (en este caso "*Bumped.class*") que ocurra, y tomar las acciones necesarias en la función definida inmediatamente después.

Navegación

Ya se conoce el modo en el que el bot obtiene información sobre su entorno y cómo es el mecanismo de recepción de estímulos. Ahora, se necesita de una concepción del entorno en el que se encuentra actualmente, la geografía del escenario, para poder llevar a cabo movimientos en coherencia con él.

El Mapa y los Puntos de Navegación

Cada mapa es una colección de datos como por ejemplo, localización de puertas, puentes, precipicios... etc., los cuales son utilizados por los objetos que trabajan por encima del mapa, es decir, los que hacen uso de la información que lo caracteriza.

Cada uno de ellos está cubierto por puntos de navegación colocados en lugares preferiblemente seguros y alcanzables por el bot. Éstos están representados por un grafo en el que cada vértice es un punto de navegación y la existencia de arcos que asocien a dos de estos vértices, significa su unión por medio de un camino directo en el mapa.

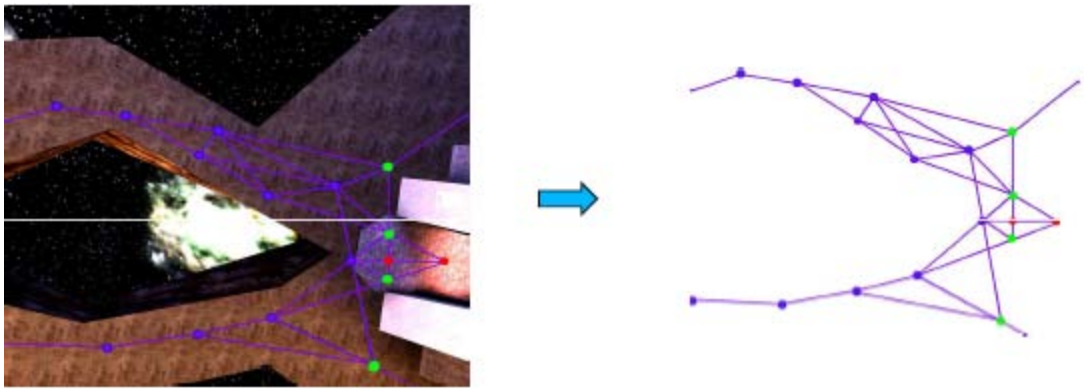


Figura 7

Los caminos y la Navegación

La interfaz *IPathPlanner* es la responsable de la planificación, valga la redundancia, del recorrido que ha de realizar el bot, dados una punto de partida y otro de llegada. Pogamut provee sendas implementaciones de la interfaz de planificación, usando algoritmos como Floyd Warshall o *A**. Para planificar un camino entre dos puntos de navegación, se utiliza la función:

```
computePath( Partida, Llegada )
```

Para realizar la navegación por un camino en concreto, se hace uso del ejecutor del camino o *IPathExecutor* el cual referencia la interfaz nombrada anteriormente, supervisando los trazos que el planificador recorre, mediante el uso de la función (entre otras):

```
followThePath( Camino )
```

Del movimiento del bot por el medio es responsable la interfaz *IUT2004PathNavigator*, la cual le permite esquivar obstáculos, abrir puertas, saltar por encima de precipicios,... etc. Esta interfaz es la parte del ejecutor de caminos más fácil de implementar, ya que sólo dispone de dos métodos:

- navigate()
- reset()

La navegación se gestiona de manera asíncrona. Así mismo, existe la posibilidad de implementar un planificador de caminos en lugar de hacer uso del que proporciona la librería por defecto.

Ejemplos de uso

A continuación se expone algunas funciones en pseudocódigo, en las que se utiliza los métodos anteriormente explicados.

El siguiente método sirve para ir en búsqueda de un ítem dado. Se debe saber la localización del objeto, computar un camino hasta dicha localización desde la que se encuentra actualmente el bot, y posteriormente ejecutarlo para que el agente comience a moverse hacia allí..

```
buscarItem(item) {
	localizacion = item.dameLocalizacion()
	caminoPlanificado = getPathPlanner().computePath(localizacion)
	getPathExecutor().followPath(caminoPlanificado)
}
```

Este es un ejemplo que se ha utilizado para el funcionamiento del sistema realizado. Básicamente, sirve para que el agente no deje de moverse en ningún momento, y que así tenga la posibilidad de recorrer el mapa encontrando a otros jugadores. Se ha de comprobar, en primera instancia, que no se esté moviendo el agente ya hacia un lugar anteriormente especificado. Dado que no tenemos un lugar fijo al que acudir, se recoge una posición aleatoria del mapa, se planifica el camino hasta allí y, posteriormente, tal y como se realizó en la búsqueda de ítem, se ejecuta.

```
merodear() {
	si noEstoyMoviendome() entonces {
		puntoNavegacion = damePuntoNavegacionAleatorio()
		caminoPlanificado = getPathPlanner().computePath(puntoNavegacion)
		getPathExecutor().followPath(caminoPlanificado)
	}
}
```

Árboles de comportamiento

Introducción

Los árboles de comportamiento son estructuras que permiten organizar el comportamiento de un sistema, enfocados al cumplimiento de objetivos. Éstos son una técnica derivada de la puesta en común de las ventajas que ofrecen las máquinas de estados finita jerárquicas (o HFSMs, del inglés Hierarchical Finite State Machines), junto con los planificadores HTNs (del inglés Hierarchical Task Networks) y las técnicas de *scripting* utilizadas en el diseño y la implementación de la IA de los agentes inteligentes en videojuegos.

Son el compromiso perfecto entre estas 3 técnicas, funcionando de manera eficaz en la práctica, siendo simple, intuitivo y fácil de analizar. Por todo ello han ido tomando bastante popularidad en el mundo del desarrollo de la inteligencia artificial.

Historia

Las primeras ideas acerca de los árboles de comportamiento y su aplicación en sistemas y en ingeniería del software, fueron desarrolladas por R.G. Dromey el cual define los árboles de comportamiento como "una estructura formal, similar a un árbol, que representa un comportamiento de una entidad individual o de una red de ellas, la cual realiza o cambia estados, toma decisiones, responde o provoca eventos y que interactúan intercambiando información y/o controlando el paso".

Los árboles de comportamiento se han convertido en herramientas de gran utilidad debido a que para realizar una implementación eficiente de sistemas software de gran tamaño, debemos capturar primero en una especificación formal de los requisitos, después en el diseño y finalmente en el software, las acciones, eventos, decisiones, y/o las obligaciones lógicas, y las constantes expresadas en los requisitos originales en lenguaje natural de un sistema. En eso son realmente útiles los árboles de comportamiento, ya que proveen una relación fácilmente "traceable" entre lo que esta expresado en lenguaje natural y su especificación en dichos árboles. La traducción de los requisitos originales en lenguaje natural a árbol de comportamiento, se realiza de forma sencilla frase a frase, o incluso palabra a palabra. Por tanto podemos decir que los árboles de comportamiento se auto describen.

Mediante un pequeño ejemplo, podemos ver la facilidad para transformar una frase en un árbol de comportamiento.

"Cuando el individuo está en la entrada, si la puerta esta abierta, el individuo entra en casa, sino el individuo coge la llave, la introduce en la puerta y la abre, entonces el individuo podrá entrar en casa"

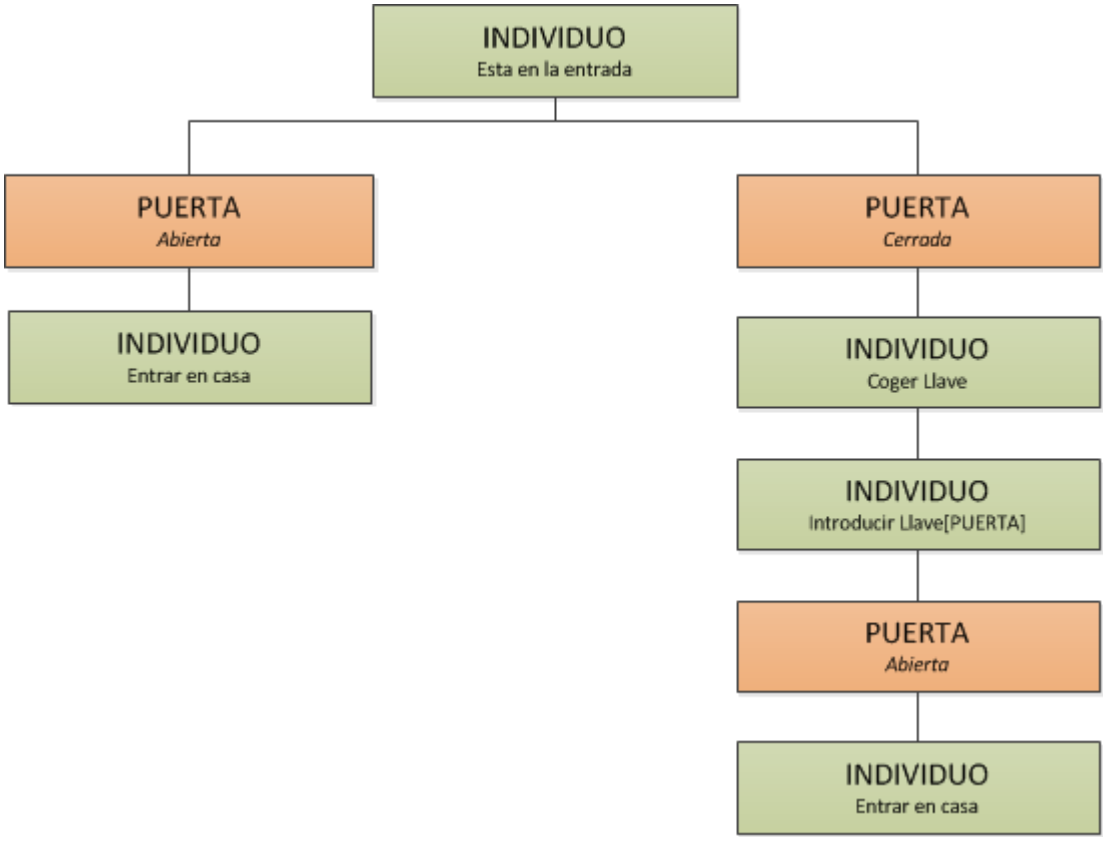


Figura 8

Esta descripción encaja con la utilidad que se le quería dar en un principio a los árboles de comportamiento, que se veían como una herramienta nueva para programadores, en vez de para diseñadores de videojuegos, los cuales adoptaron dichos árboles de comportamiento, para el desarrollo de agentes inteligentes en videojuegos.

El ejemplo más representativo lo encontramos en el desarrollo del videojuego Halo 3, en el cual pasaron del uso de métodos imperativos basados en FSMs al uso de árboles de comportamiento (BTs) para el control de los agentes inteligentes. Éstos, por el contrario, siguen

una metodología declarativa, ya que se enumeran las tareas que se necesita realizar en el entorno.

También es de interés mencionar el trabajo de Michael Mateas y Andrew Stern llamado *Façade*, el cual es una historia interactiva entre el usuario y personajes cuyo objetivo es aplicar distintos diálogos e interactuar con el jugador. Aquí son utilizados árboles de comportamiento formales, por lo que muestra su potencia y flexibilidad al poder ser aplicados en tipos de videojuego distintos al de primera persona.

Componentes

Un árbol de comportamiento está compuesto por nodos y hojas, tal como cualquier árbol lo está, pero cada uno de ellos tiene una finalidad diferente como se verá a continuación.

En un árbol de comportamiento, cada nodo representa un comportamiento o tarea, las cuales salvo casos excepcionales, estarán compuestos a su vez de otros comportamientos o sub-tareas más sencillas. Poniendo como ejemplo cuando un jugador quiere atacar, éste se mueve, apunta al enemigo y dispara. El hecho de que cada tarea esté compuesta de diferentes tareas resulta realmente útil, ya que permite la encapsulación de comportamientos.

Un ejemplo muy sencillo se puede apreciar en la figura siguiente, en la que se representa una tarea compuesta por cuatro nodos hoja que, en este caso, son tareas de bajo nivel (interactúan con el motor del juego de forma directa).

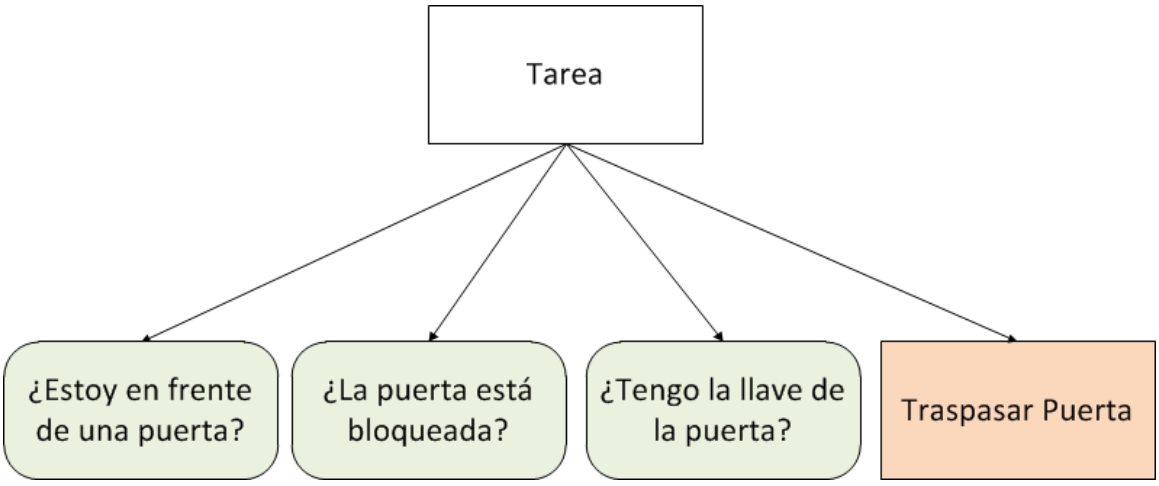


Figura 9

Tareas de bajo nivel

Tal y como se ha indicado anteriormente, éstos son las tareas que se comunican directamente con el motor del juego, los cuales se sitúan en nodos hoja. Éstas son de dos tipos fundamentalmente:

Acciones

Las acciones son aquellas que implementan una acción del actor del juego, o un cambio en el estado del juego o del mundo donde se desarrolla el juego, como sentir enemigos cerca, correr en una dirección planeada, elegir arma, etc..

Condiciones

Las condiciones son aquellas cláusulas que han de cumplirse para cierto actor en el estado del juego, para que continúe con la secuencia de tareas que la suceden en el nodo. Véase por ejemplo, que para que un jugador dispare a un enemigo con el arma que tiene actualmente, este arma ha de cumplir la condición de que aún le quede munición, en caso contrario la condición no se cumplirá y la acción no será realizada, realizándose en su lugar otra que cumpla sus condiciones.

Además de estar compuesto de acciones y condiciones, existen también tareas compuestas por diferentes estructuras, la cual nos hace diferenciar entre varios tipos de nodos compuestos, siendo las más importantes los nodos secuencias, los nodos selectores y los nodos de ejecución en paralelo.

Nodos Compuestos

Nodos secuencia

Estos nodos están compuestos de un grupo de tareas que se van ejecutando una tras otra, en orden de izquierda a derecha, mientras ninguna condición falle. Pueden estar compuestos de una tarea simple, por ejemplo, si tenemos una condición “puedo Atacar” que se cumple, la acción a realizar sería atacar, o tareas más complejas, compuestas de varias condiciones y varias acciones. El nodo secuencia se ejecutará con éxito si todas las tareas que lo componen se ejecutan con éxito.

Desde el punto de vista lógico, el nodo secuencia se ejecuta como una puerta AND, si uno falla, falla todo el nodo secuencia.

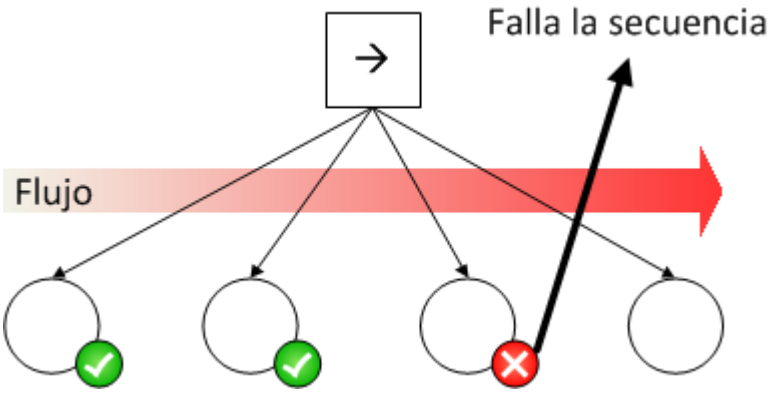


Figura 10

Nodos selector

Estos nodos son bastante parecidos a los nodos secuencia, ya que también se componen de un grupo de tareas que se ejecutan una tras otra, de izquierda a derecha, si bien cuando una de las tareas a realizar dentro del nodo resulta exitosa, el nodo selector finalizará en éxito también. En caso contrario, continuará ejecutando las demás tareas hasta el encontrar el primer éxito. En caso de que ninguna de las tareas tenga éxito, el nodo selector fallará.

Desde el punto de vista lógico, este nodo actúa como una puerta OR en la que, si una tarea tiene éxito, el nodo selector tiene éxito.

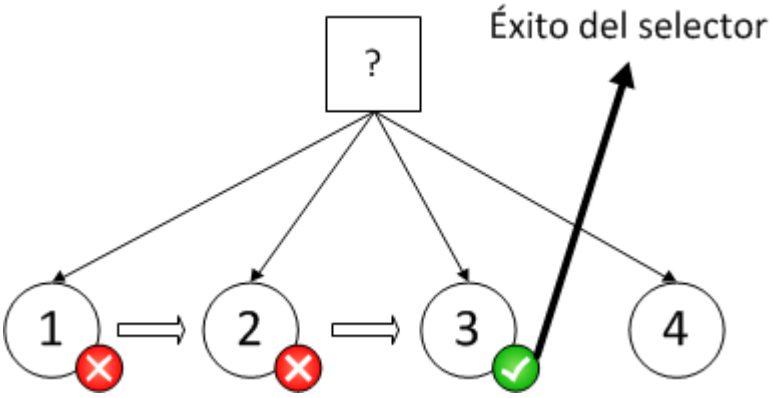


Figura 11

Nodos de ejecución en paralelo

Estos nodos están compuestos de un grupo de tareas, las cuales son ejecutados en paralelo. Su ejecución depende de la política en la que se base. Ésta puede ser política secuencial (funcionando como los nodos secuencia) o de selector (funcionando análogamente a los nodos selector).

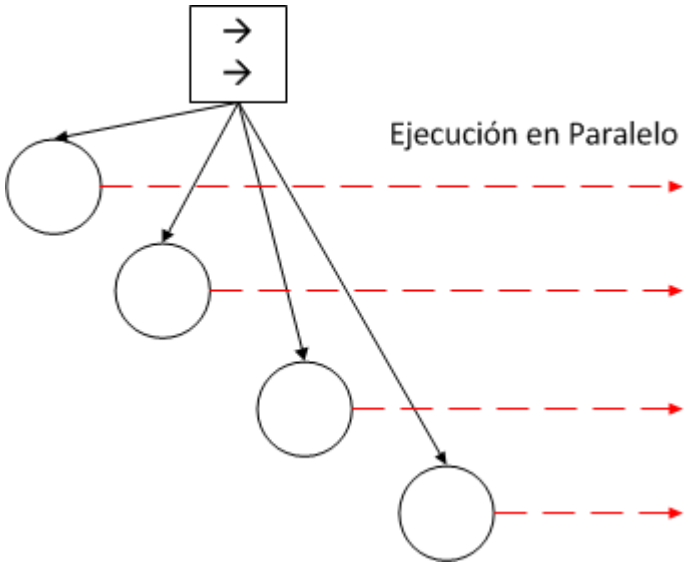


Figura 12

Así mismo, disponemos de otro tipo de nodos, el cual nos da la posibilidad de encapsular y cambiar el comportamiento de tareas. Éstos son los decoradores.

Decoradores o decorators

Son aquellos los cuales, como bien indica su nombre, "decoran" el nodo que cuelga de ellos. Esto es muy parecido a lo que ocurre en programación, cuando se hace uso del patrón de diseño *Decorator*. Éste añade dinámicamente funcionalidad a un objeto, permitiendo no tener que crear sucesivas tareas para llevarlo a cabo.

Como ejemplo, el decorador *Inverter*, lo que hace es invertir el valor de retorno de la tarea a la que se aplica, cambiándola de éxito a fallo y viceversa. .

Finalmente, cabe destacar la existencia de otros tipos de nodos hoja, como son las referencias a subárboles, cuya finalidad es la de encapsular árboles de comportamiento enteros.

Todos estos componentes pueden ser combinados y puestos en marcha para así realizar árboles de mayor profundidad y complejidad, que puede mostrar la especificación de grandes sistemas software, o el comportamiento de determinados agentes inteligentes dentro de un videojuego.

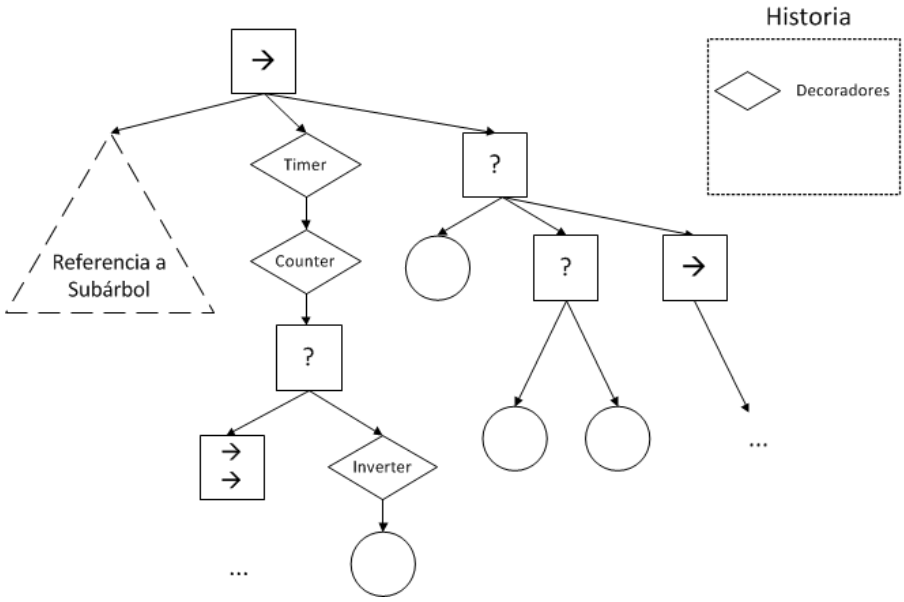


Figura 13

Ventajas

Una de las propiedades más interesantes de los árboles de comportamiento es su capacidad de reutilización. A diferencia de otras técnicas donde la reutilización es complicada, como las FSMs, la misma naturaleza de los árboles de comportamiento hace posible que sea fácil reutilizarlos en cualquier situación.

Otra de las principales ventajas que se encuentra es que, para realizar su representación la notación es simple y sencilla, siendo fácil especificar el comportamiento del sistema: tanto las condiciones que han de cumplirse en cada nodo como las acciones a realizarse. Esto hace que los árboles de comportamiento sean fácilmente representables y, por tanto, la complejidad del diseño se vea reducida significativamente. Esta reducción afecta positivamente también a uno de los problemas habituales en el desarrollo de software, el relacionado con la sobrecarga de memoria.

Éstos aportan una solución al problema derivado del crecimiento exponencial de los estados de transición en una máquina de estados finita (FSM), gracias a una definición más restrictiva pero a su vez más estructurada.

Cabe también referenciar la versatilidad que presenta en el mantenimiento de errores. Al no trabajar con líneas interminables de código, sino con representaciones abstractas e intuitivas la depuración del sistema se hace mucho más sencilla.

Ejemplos

Sistemas biológicos

Debido a que los árboles de comportamiento describen un comportamiento complejo, pueden ser utilizados para describir un rango de sistemas que no estén limitados a aquellos basados en computación. En un contexto biológico, los BTs pueden ser utilizados para unir una interpretación procedural de funciones biológicas descritas en investigaciones, tratando los documentos como requisitos para el sistema. Esto puede ayudar a construir una descripción más concreta del proceso que la que sería posible sólo leyendo los documentos que la componen, pudiendo ser utilizados para comparar teorías que tratan acerca de los mismos fundamentos. En investigaciones actuales, la notación de los árboles de comportamiento está siendo utilizada en animales para el desarrollo de modelos de funciones del cerebro bajo condiciones extremas.

Control de acceso basado en roles

Para asegurar la correcta implementación de los requisitos de accesos de control complejos, es importante que los requisitos validados y verificados sean integrados satisfactoriamente con el sistema. Es también de importancia que el sistema global sea validado y verificado en el proceso de desarrollo (fase posterior a la colección de requisitos). El modelo del control de acceso basado en roles, está basado en la notación de los árboles de comportamiento, que pueden ser validados en simulaciones así como verificados, utilizando comprobadores aplicados al árbol de comportamiento en sí.

Por consiguiente, haciendo uso de este modelo, los requisitos para el control de acceso pueden ser integrados con el resto del sistema de manera efectiva y correcta.

Halo 3

Como ya se comentó anteriormente, el caso de Halo 3, sentó las bases del uso de BTs en el ámbito de la inteligencia artificial orientada a videojuegos. El problema emergió debido al crecimiento de entidades independientes en el sistema, el aumento del tamaño de los escenarios y la mayor combinación de movimientos posibles. Con FSMs como herramienta aplicada en la inteligencia artificial de los agentes, a medida que la complejidad del sistema crecía, la complejidad lo hacía en orden cuadrático al número de transiciones entre aristas. La nueva estrategia fue pasar de ese método imperativo a uno declarativo, enumerando las tareas que se necesitaban hacer para interactuar con el medio, y dejar libertad al sistema para determinar quién debía llevarlas a cabo. Así, aparte de reducir dicha complejidad, se ahorraron multitudes de líneas de código *script* (enfocadas a la inteligencia artificial del agente) enfocándolas a acciones a ser activadas por los BTs (mínima lógica relacionada con comportamiento en código).

Una visión muy general del árbol de comportamiento que controla el funcionamiento del bot del juego la encontramos en la siguiente imagen.

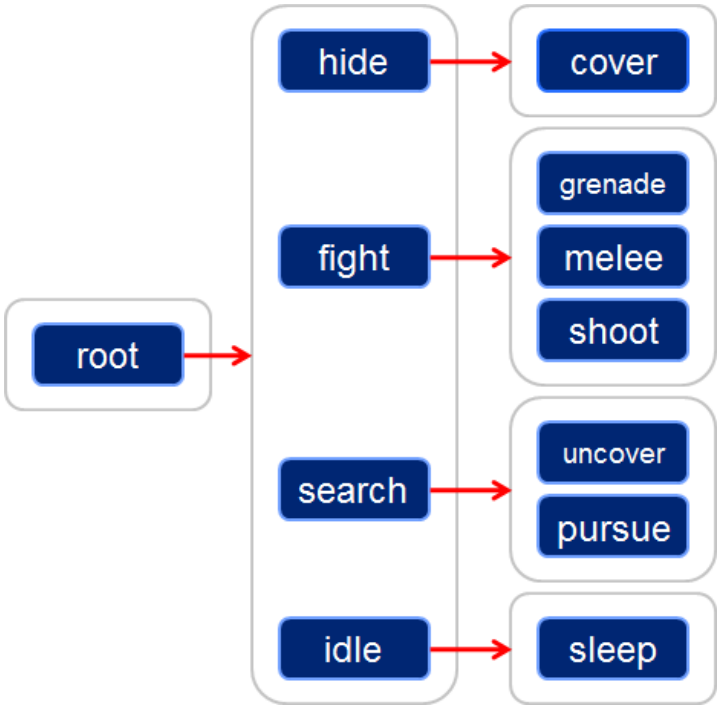


Figura 14

Bot con Árboles de comportamiento

Librería utilizada: JBT (Java Behaviour Trees)

JBT es un *framework* de Java para construir y ejecutar árboles de comportamiento. Tiene dos partes principales. Por un lado, encontramos el núcleo JBT Core, que implementa todas las clases necesarias para crear y ejecutar un árbol de comportamiento. Para facilitar la tarea de creación de árboles de comportamiento, ésta incluye herramientas para automatizar el proceso de creación. En particular, permite crear código Java directamente desde la descripción del árbol en código XML. Por este motivo, el usuario sólo ha de preocuparse por la definición de estos árboles en XML y de la implementación de las acciones y condiciones de bajo nivel que el árbol va a utilizar, implementadas también en XML, formando lo que se denomina dominio del árbol. Este dominio es una definición conceptual, siguiendo el formato de MMPM (Make Me Play Me), una arquitectura que facilita la conexión entre estrategias de juego y técnicas de inteligencia artificial, en nuestro caso BTs.

Por otra parte esta librería, permite al usuario crear árboles de comportamiento por medio de un editor que permite ir añadiendo nodos, condiciones, acciones, etc., como se muestra a continuación **[Figura 15]**.

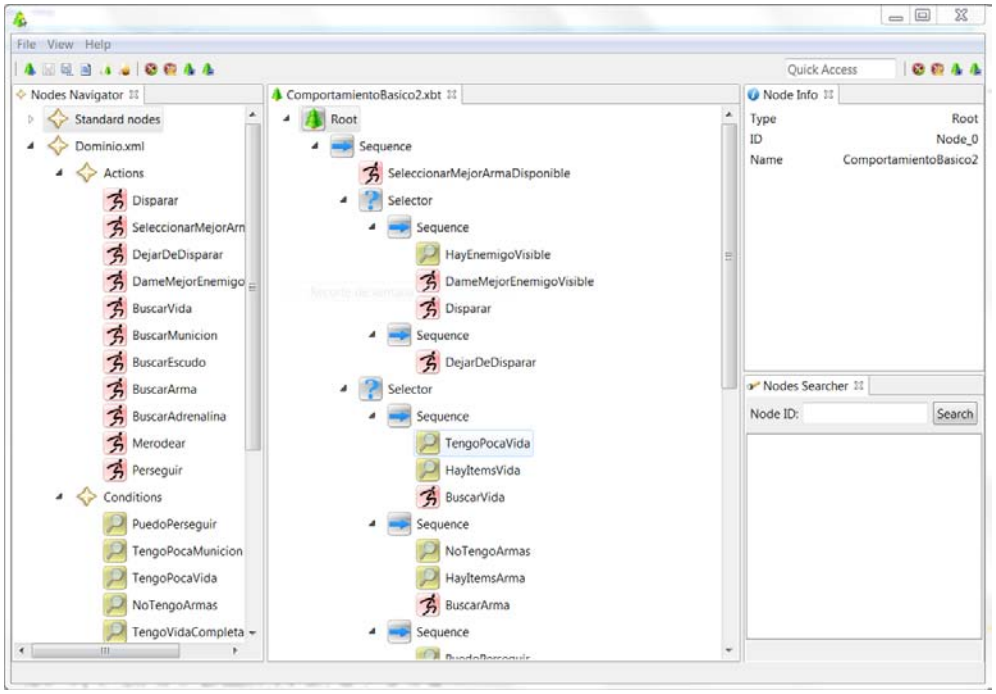


Figura 15

Componentes utilizados

A continuación, se listan los componentes que han sido utilizados en el desarrollo del árbol de comportamiento para el desarrollo del agente con JBTs:

-  Selectors: Tarea que ejecuta todos los nodos hijo en orden. Si uno de ellos tiene éxito, entonces la tarea global se detiene y tiene éxito.
-  Sequences: Tarea que ejecuta todos los nodos hijo en orden. Si uno de ellos falla, entonces la tarea global se detiene y falla.
-  Random Selectors: El comportamiento es idéntico al de su homónimo no aleatorio, salvo que los nodos hijos se ejecutan de forma aleatoria.
-  Random Sequences: El comportamiento es idéntico las secuencias, pero los nodos hijo se ejecutan en orden aleatorio.
-  Conditions: Condición genérica que es ejecutada en el juego.
-  Actions: Acción genérica que se ejecuta en el juego.
-  Inverters: Tarea utilizada para invertir el valor de retorno devuelto por su nodo hijo. Cuando la tarea a la que se aplica este inversor finaliza, el valor de retorno se invierte, es decir, si tiene éxito devuelve fallo y viceversa.
-  Tree Lookups: Nodo que sirve como puntero que referencia a otro árbol de comportamiento, el valor de retorno de este nodo, es el valor de retorno del árbol al que apunta.
-  Succeders: Tarea que siempre tiene éxito.
-  Failures: Tarea que siempre falla.

Diseño del árbol

Como toma de contacto con la librería, se realizó el árbol básico de la siguiente figura.. Aquí el bot comprueba, con la condición *"PuedeAtacar"* **A**, si se cumplen una serie de premisas (si hay enemigos visibles, si se poseen armas cargadas...), las cuales son imprescindibles para que el bot lleve a cabo las acciones siguientes, en las que se selecciona el mejor enemigo visible al que se pueda atacar **B,** seguidamente se selecciona el mejor arma dado el enemigo seleccionado anteriormente **C** y finalmente se lleva a cabo el ataque **D**.

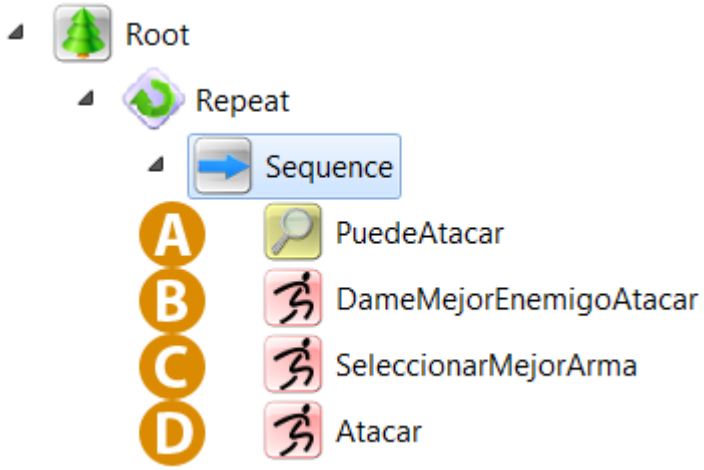


Figura 16

Para el diseño del árbol se ha utilizado el modelo descrito a continuación:

Antes de cualquier acción, se selecciona la mejor arma disponible en el inventario del bot (**A**), para posteriormente realizar las tareas **B** y **C** las cuales serán detalladas a continuación.

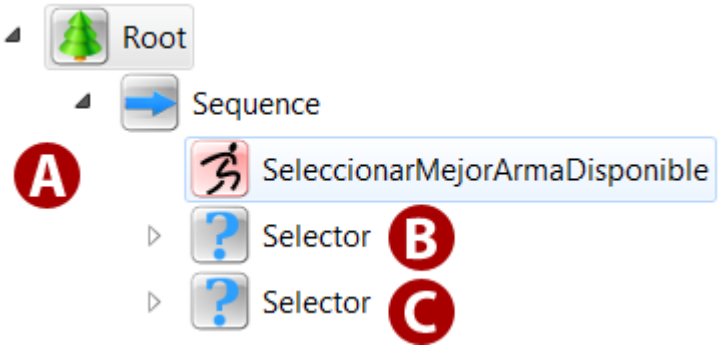


Figura 17

La primera de las tareas, es la que se puede apreciar en la figura **[Figura 18]**. Si hay algún enemigo visible (**A**), se selecciona al mejor de todos (**B**), esto es, al que sea mejor atacar, y posteriormente el agente dispara al enemigo seleccionado (**C**). En caso de que no haya ningún enemigo visible, el agente cesa de disparar (**D**).

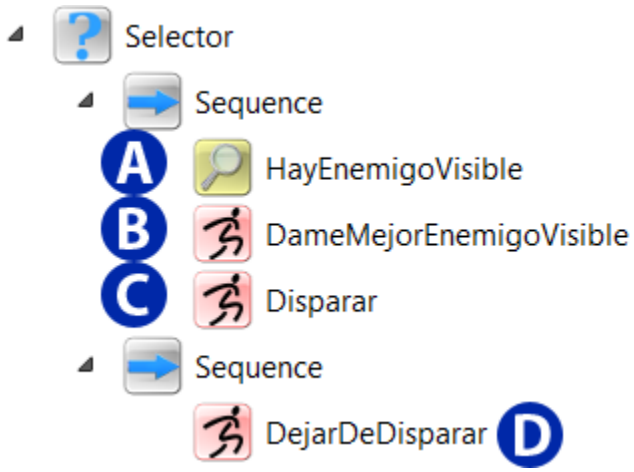


Figura 18

La segunda tarea es un tanto más compleja. En primera instancia tenemos la acción a la que se ha dado más importancia en el árbol de comportamiento, la cual es ir en búsqueda de vida. Como se puede apreciar en la figura **[Figura 19]**, se comprueba si el nivel de vida del agente es inferior a un mínimo establecido (**A**), posteriormente se observa si existen ítems de vida disponibles (**B**) y finalmente se va en busca de vida (**C**). Este último chequeo se realiza para asegurar al navegador que sí hay ítems en el escenario y así evitar que realice la búsqueda cuando no es posible. Este razonamiento se ha seguido para todas las tareas de búsqueda que se aprecian en la figura indicada. Después de la tarea de búsqueda de vida, se encuentra las tareas: búsqueda de armas, perseguir (cuyo único propósito es el de ir detrás del enemigo seleccionado anteriormente en el árbol, para poder así atacar de forma más efectiva), buscar munición y por último, dos tareas (**F** y **G**) las cuales explicamos a continuación.

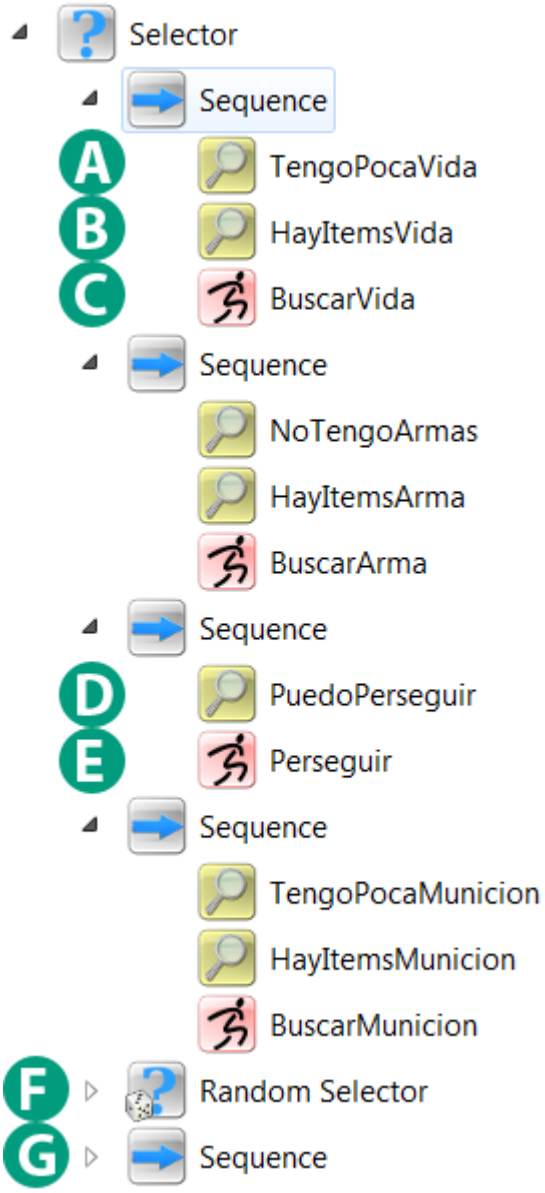


Figura 19

Dado que buscar escudo y buscar adrenalina, son tareas que se han considerado de baja prioridad en el comportamiento del bot, han sido incorporadas en un selector aleatorio, como se muestra en la figura **[Figura 20]**. Por último, cuando no se cumplen las condiciones para llevar a cabo búsquedas de ningún tipo de ítem, se encuentra la tarea de merodear (**F**), en la que, como bien se ha explicado en la sección de navegación, el bot se va moviendo por puntos aleatorios del mapa.

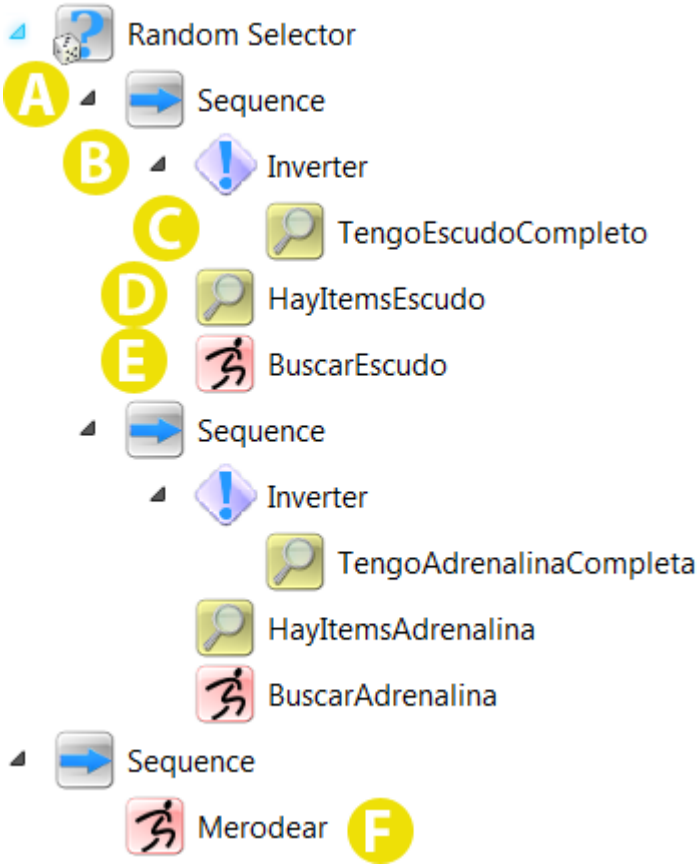


Figura 20

Integración con el bot

En este apartado se detalla la forma de integrar un árbol de comportamiento generado por JBT, a un proyecto java de UT2004.

Lo que hemos visto hasta ahora detalla el proceso de diseñar de manera gráfica un árbol de comportamiento con JBT, pero con este diseño solamente no conseguimos mucho. Ahora es el turno de generar el código que nos permitirá ejecutar dicho comportamiento en un entorno real de ejecución.

JBT nos proporciona unas herramientas para generar código (a partir del diseño gráfico que previamente hemos desarrollado) en clases y modelos de un lenguaje como Java. Éstas consisten fundamentalmente en los siguientes ficheros:

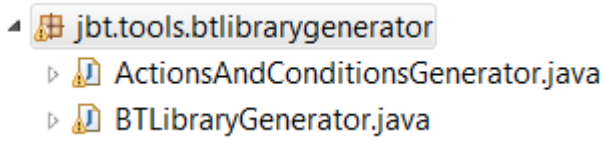


Figura 21

Para generar las acciones y condiciones, se ha de proporcionar un archivo de configuración como parámetro al ejecutar el archivo *ActionsAndConditionsGenerator.java*. En este archivo de configuración indicamos los paquetes en los que queremos englobar tanto el modelo del árbol como la parte de ejecución. Análogamente, para generar el código referente a la estructura y organización del árbol de comportamiento, se ha de ejecutar el archivo *BTLibraryGenerator.java*, pasándole como parámetro de ejecución un archivo de configuración en el que se indica, entre otras cosas, el paquete en el que queremos que resida el fichero de implementación del árbol.

Después de generar el código podemos ver que tenemos una estructura con varios paquetes como la siguiente:

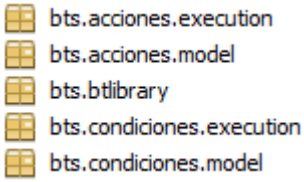


Figura 22

En esta estructura encontramos los modelos y los ejecutores que se utilizan para emular el comportamiento diseñado en el árbol.

En el paquete bts.btlibrary encontramos una sola clase en la que vemos como se incluyen todas las acciones y condiciones que hemos descrito. De hecho con el código bien formateado se parece bastante al diseño gráfico.

Para que los ejecutores funcionen hay que decirles qué función deben de llamar de nuestro bot. Por ejemplo la acción atacar dibujada en el árbol puede ser una función llamada disparar en nuestro bot. Tenemos entonces que modificar las clases que se encuentran en "bts.XXXX.execution" para integrar nuestro código y hacer así que el bot obtenga la inteligencia del árbol de comportamiento.

Antes de nada detallaremos cómo y dónde podemos inicializar el árbol cuando estamos creando el bot.

Como hemos explicado al principio de este documento la función “prepareBot()” es un método que inicializa las variables del bot, antes de que éste empiece a interactuar con la partida. Pues bien, esta función es la idónea para crear toda la estructura necesaria para el árbol de comportamiento.

Primero creamos la instancia del árbol, esta clase recibe el nombre con el que hayamos guardado el árbol. Esta clase se encuentra en “bts.btlibrary”:

```
var arbolDeComportamiento = new ComportamientoBasicoBTLibrary();
```

Después necesitamos crear un contexto para guardar la información adicional que necesitamos pasar como puede ser la instancia del bot para llamar a sus métodos desde las clases del árbol.

```
contextoArbol = ContextFactory.createContext(arbolBTLibrary);
contextoArbol.setVariable("bot", this);
modeloArbol = arbolBTLibrary.getBT("ComportamientoBasico");
```

Con esto ya tenemos el árbol inicializado, ahora tenemos que introducirlo en el bucle principal del juego para que evalúe siempre el árbol en cada iteración y ejecute siempre la mejor acción posible.

Para ello debemos de introducir el siguiente código en la función “logic()” del bot que es la que se ejecuta siempre en cada tick del juego (el bucle principal):

Creamos el ejecutor pasándole el modelo y el contexto, antes hemos visto como crear el contexto pero no cómo generar el modelo.

Basta con pedírselo al árbol.

```
modeloArbol = arbolBTLibrary.getBT("ComportamientoBasico");
```

Luego, ya podemos crear el ejecutor.

```
executorAcciones = createBTExecutor(modeloArbol, contextoArbol);
```

Lo metemos en un bucle para asegurarnos que recorre todo el árbol hasta que termine su ejecución.

```
do {
      executorAcciones.tick();
      }
while (executorAcciones.getStatus() == Status.RUNNING);
```

Después de tener el árbol ya preparado aunque ejecutemos el bot vemos que no hace nada, no se mueve, no dispara. Eso es porque las clases encargadas de llamar a los métodos que hacen eso posible aún no están integradas.

Estás clases (“bts.XXXX.execution.xxxx.java”) están compuestas por las siguientes funciones:

- constructor()
- internalSpawn()
- internalTick()
- internalTerminate()
- restoreState()
- storeState()
- storeTerminationState()

La función más interesante es “internalSpawn”, puesto que es donde se ejecuta la condición o acción que corresponda según el árbol. Lo primero que debe hacer esta función es coger la variable de la clase del bot que guardamos en contexto para poder acceder a los métodos (acciones y condiciones).

```
var bot = ((BasicBot) this.getContext().getVariable("bot"));
```

Después ya podemos llamar al método que corresponda, por ejemplo una condición “puedoAtacar()”. En el caso de las condiciones tenemos que guardar en el contexto su resultado:

```
this.getContext().setVariable("puedoAtacar", resultado);
```

La siguiente función que deberemos modificar es el tick interno. En ella devolveremos el resultado de la condición o de la acción. Por ejemplo “SUCCESS” si puedo atacar o “FAILURE” si la condición ha fallado y no puedo atacar.

```
if (this.getContext().getVariable("puedoAtacarr").equals(true)) {
    return jbt.execution.core.ExecutionTask.Status.SUCCESS;
}
```

```
return jbt.execution.core.ExecutionTask.Status.FAILURE;
```

Ahora si ejecutamos el bot ya debería de moverse y actuar según la lógica diseña en el árbol de comportamiento.

CBR

Introducción

"Razonamiento basado en casos es [...] la manera en la que los seres humanos usan sus experiencias para actuar, y también la manera en la que hacemos que las máquinas las usen." - Kolodner, 1993.

El razonamiento basado en casos, o CBR (siglas provenientes del inglés que significan *Case Based Reasoning*), es una técnica de inteligencia artificial basada en la recopilación en memoria de situaciones experimentadas en el pasado (casos), para resolver nuevos problemas que se van presentando. Este proceso se desarrolla como sigue:

- Se extrae la experiencia sobre situaciones similares de memoria (búsqueda y extracción).
- Ésta es reutilizada y aplicada en el contexto de la nueva situación, extrayendo y utilizando parcial o totalmente la solución (adaptación).
- Se almacena la nueva experiencia con la solución aplicada en memoria (aprendizaje).

Se le denomina sistema basado en conocimiento, entre otras razones porque los procesos que caracterizan a esta técnica, se pueden ver como una réplica de un tipo particular de razonamiento humano. En muchas situaciones, los problemas que los seres humanos encontramos son resueltos con un método CBR equivalente. Cuando una persona encuentra una nueva situación o problema, por lo general, se fijará en situaciones o problemas vividos anteriormente. Esta experiencia previa puede ser propia o "conseguida" de otra persona, la cual habrá sido añadida a la memoria del ser razonador previamente, por medio oral, escrito, etc.

Esta técnica es bastante intuitiva y familiar al ser humano desde el aspecto lógico, ya que es intrínseco al propio ser que razona y argumenta, valorando en mayor medida las vivencias adquiridas a lo largo de su existencia al momento de tomar decisiones cualesquiera.

Enfocado desde una perspectiva computacional, su aplicación es también intuitiva, al contrario de otros sistemas basados en conocimiento como se verá más adelante.

Un ejemplo de otro sistema de estas características es el sistema basado en reglas, que consiste en un conjunto de condiciones de la forma "*si* condición *entonces* acción". El número de estas condiciones puede crecer según el ámbito en el que nos movamos, y depende de uno mismo el adquirir el conocimiento simbólico que está representado en ese conjunto de reglas/condiciones, por medio de métodos manuales o automatizados según el caso. En algunas ocasiones, se utiliza un modelo del problema como base para razonar acerca de una situación, pasando a denominarse sistemas basados en modelos.Una de las tareas que consumen más tiempo cuando se trata con un sistema basado en reglas es la adquisición del conocimiento. Adquirir información específica de un dominio y transformarla en la representación formal, puede ser inadmisible en algunos casos, especialmente cuando el contexto del dominio es muy abstracto o poco investigado.

Los sistemas basados en casos requieren menor adquisición de conocimiento previo incluso en los dominios anteriores. Éstos pueden ser desarrollados incrementalmente añadiendo casos a la base del conocimiento a medida que se vaya ampliando el aprendizaje sobre el campo, y haya más experiencias que sirvan de ejemplo.

Como se verá más adelante, esta técnica puede aplicarse a campos tan diversos como la refutación de argumentos, la selección y análisis de datos de microarrays, la detección de intrusos a nivel servidor, detección de spam, diagnosis de enfermedades en sistemas expertos aplicados a la medicina, etc.

Historia

Las primeras ideas acerca del "razonamiento basado en casos" las podemos encontrar en el trabajo de Roger Schank y sus estudiantes en la universidad de Yale a principio de la década de los 80, el cual define el modelo de memoria dinámica. Dicho modelo es la base de los primeros sistemas CBR.

Durante la décadas de los 80 aparecen varias vías de investigación dentro del Razonamiento basado en casos, tales como el razonamiento legal, el razonamiento basado en la memoria, y combinaciones de razonamiento basado en casos con otros métodos de razonamiento.

En los años 90 se estableció una conferencia internacional sobre el razonamiento basado en casos, así como en algunos países como Alemania, Gran Bretaña, Italia, etc. Se produjeron numerosos casos de éxito como el sistema llamado "Lockheed´s CLAVIER".

También está siendo utilizado con frecuencia en aplicaciones de ayuda de escritorio como “Compaq SMART system”.

El Razonamiento Basado en casos ha experimentado un rápido crecimiento en los últimos años, desde su nacimiento en Estados Unidos. Lo que sólo parecía interesante para un área de investigación muy reducida, se ha convertido en una materia de amplio interés, multidisciplinar y de gran interés comercial.

Las aplicaciones CBR se clasifican principalmente en dos tipos: tareas de clasificación y tareas de síntesis.

En las tareas de clasificación, un caso nuevo se empareja (matching en su terminología original) con aquellos de la base de casos para determinar qué tipo, clase o caso es. La solución del caso que mejor ajusta es el que se reutiliza.

La mayoría de las herramientas CBR disponible dan un soporte aceptable para las tareas de clasificación, que suelen estar relacionadas con la recuperación de casos. Existe una gran variedad de tareas de clasificación, como por ejemplo:

- Diagnosis: Médica o de fallos de equipos.
- Predicción: Pronóstico de fallos de equipos o actuación sobre el stock de un mercado.
- Valoración: análisis de riesgos para bancos o seguros o estimación de costes de proyectos.
- Control de procesos: Control de fabricación de equipos.
- Planificación: Reutilización de planos de viaje o planificadores de trabajo.

Las tareas de síntesis intentan crear una nueva solución combinando partes de soluciones previas. Éstas son inherentemente complejas a causa de las restricciones de los elementos usados durante la síntesis.

Los Sistemas CBR que realizan tareas de síntesis deben realizar adaptación y son normalmente sistemas híbridos que combinan CBR con otras técnicas.

Tareas de clasificación

Las tareas de clasificación son habituales en el mundo de los negocios. Se da cuando se necesita emparejar un objeto o evento con otros en una librería en la cual se puede inferir una respuesta.

Algunos ejemplos ilustrativos de éstas en el ámbito de los negocios son:

- ¿Qué riesgo tiene un cliente usuario de una línea de crédito?
    - Alto.
    - Medio.
    - Bajo.
- ¿Existe la posibilidad de ahorrar dinero en España?
    - Posible.
    - Imposible.
    - Probable.
    - Improbable.
- ¿Qué tipo de coche es este?
    - Deportivo.
    - Todo terreno.
    - Caravana.
- ¿Con qué diagnosticar a un paciente?
    - Antibióticos.
    - Antihistamínicos.
    - Diuréticos.

Estos valores pueden estar definidos en ciertos rangos.

Habitualmente, las clasificaciones de las preguntas se refieren a resultados, esto es, un resultado es normalmente un atributo del caso.

Podemos aplicar CBR fácilmente a problemas de clasificación puesto que pueden consistir en:

- La recuperación de un amplio conjunto de casos similares, por ejemplo aquellos en los que el antibiótico fue el tratamiento.
- Recuperar el mejor ajuste de este conjunto, quizás para sugerir penicilina como antibiótico específico.
- Adaptar la solución, por ejemplo alterando la dosis para diferentes edades o pesos de los pacientes.
- Almacenar el resultado del nuevo caso para un futuro uso.

Como vemos, las tareas de clasificación son fáciles de implementar porque se ajustan al ciclo CBR, los casos tienden a ser más fáciles de representar y recuperar, y los algoritmos de recuperación utilizados en la mayoría de las herramientas CBR son clasificadores.

Tareas de síntesis

Estas tareas son comunes en el comercio pero difíciles de implementar. Esto es debido a la facilidad de ajustar un artefacto a otro conjunto de artefactos que construir un artefacto a partir de una especificación.

Las tareas de síntesis requieren colocar las características en el orden y lugar correcto mientras que las tareas de clasificación sólo requieren reconocer las características.

Los sistemas de síntesis mediante un diseño o planificación simplifican el proceso creativo produciendo un diseño o plan que resulta de utilidad para producir el plan final a partir de él. Esto es más rápido que empezar un diseño desde una hoja en blanco ya que modificar un buen diseño o plan inicial es más fácil que crear uno desde el principio.

Sin embargo hay muchas situaciones en que se debe empezar desde cero sin tener referencia de ningún ejemplo pasado. Por ejemplo, cuando se realiza el cambio de tecnología a utilizar en los microchips (porque se introducen mas núcleos por ejemplo), se comienza el diseño desde cero, sin basarse en los modelos anteriores.

Las razones por las que los sistemas de síntesis son difíciles de construir son:

- La representación de un caso de un diseño es compleja y altamente estructurada con muchas dependencias entre características. Los casos no se almacenan en un medio único y homogéneo, por tanto la recuperación de casos es más difícil.
- Las herramientas CBR difícilmente pueden recuperar representaciones de casos altamente estructurados.
- La adaptación es a menudo un requisito clave en las tareas de síntesis

Componentes CBR

¿Qué es un caso?

Figura 23

"Un caso es un fragmento contextualizado de conocimiento que representa una experiencia y que enseña una lección importante para conseguir los objetivos del razonador" [Kolodner & Leake 97].

Podemos considerar un caso como un acontecimiento o experiencia previa a un problema. Las características que se registran sobre un acontecimiento depende tanto del dominio como del propósito para el que se use el caso. La descripción del problema debe incluir:

- Los objetivos necesarios para resolver el problema.
- Las restricciones de los objetivos.
- Las características de la situación del problema y las relaciones entre sus partes.

Otra información muy importante que se debe almacenar es la descripción de la solución, que será usada cuando nos encontremos en una situación similar. Esta descripción puede incluir únicamente los hechos que llevan a la solución o información sobre pasos adicionales en el proceso de obtención de la solución. Además esta descripción de la solución también podrá incluir:

- Las decisiones tomadas para la obtención de la solución.
- Soluciones alternativas que no fueron elegidas y el porqué de dicho descarte.
- Soluciones rechazadas y su porqué.
- Expectativas acerca del resultado de la solución.

También es importante incluir una medida del éxito de las soluciones con diferentes niveles de éxito o fracaso.

Además también se puede incluir información acerca del resultado:

- Si el resultado cumple o no con las expectativas.
- Si el resultado fue un éxito o un fracaso.
- Qué podría haberse hecho para evitar el problema
- Un puntero al siguiente intento de solución.

El conocimiento de un caso siempre deberá ser específico, es decir, el conocimiento debe de ser concreto a una circunstancia o problema dado. En general todo el conocimiento relacionado a un mismo problema o similar se encuentra agrupado en unos pocos casos.

En los sistemas CBR la base de casos es la memoria de casos almacenados. A la hora de construir la memoria debemos tener en cuenta los siguientes aspectos:

- La estructura y representación de los casos.
- El modelo de memoria usado para organizar los casos.
- Los índices empleados para identificar cada caso.

Cómo representar un caso

Los casos pueden representar distintos tipos de conocimiento y pueden ser almacenados en diferentes formatos. Esto dependerá del tipo de sistema CBR, por ejemplo los casos pueden representar personas, objetos, diagnósticos, planes, etc.

En muchas aplicaciones prácticas los casos se representan como dos conjuntos de pares atributo-valor que representan el problema y las características de la solución.

Sin embargo, es muy difícil decidir exactamente qué representar, ya que en muchos ámbitos es muy complicado definir el dominio en el que nos manejamos. Y un mismo caso puede tener muchas representaciones dependiendo de aquello a lo que queremos enfocarlo.

Un ejemplo lo podemos encontrar en la venta de una casa. Podemos hacer una base de casos haciendo referencia a sus ventas anteriores, y el precio al cual ha sido vendido, o por otra parte hacer una base de datos con las características propias de la casa, como si esta amueblado el salón, si la calefacción está centralizada o funciona por radiadores.

Una de las ventajas del razonamiento basado en casos es la flexibilidad que ofrece respecto a la representación. Se puede elegir la implementación adecuada dependiendo del tipo de información a representar, variando desde información cualitativa, cuantitativa, específica o abstracta.

A la hora de elegir una representación para un caso se deben tener en cuenta su estructura interna, el tipo y estructura de información que va a representar, el lenguaje en el que se va a implementar el sistema, el mecanismo de búsqueda e indexación que se ha de emplear y la forma en que los casos son obtenidos.

Independientemente de la representación que elijamos siempre debemos tener en cuenta que la información que almacene un caso debe ser relevante tanto para el propósito del sistema como para asegurar que siempre será elegido el caso más apropiado para solucionar un nuevo problema en un determinado contexto.

Una vez elegida la representación de los casos, la elección del modelo de memoria es el siguiente paso. Existen principalmente dos estructuras de memoria, plana y jerárquica.

En una estructura plana de la base, los casos se almacenan secuencialmente, mientras que en una estructura jerárquica los casos se agrupan en categorías para reducir el número de casos a buscar en una consulta.

Dependiendo del método de búsqueda que apliquemos a estas dos estructuras obtenemos una serie de estructuras de memorias derivadas que explicaremos a continuación.

Utilizaremos el siguiente ejemplo para ilustrar las estructuras de memoria: supongamos que nuestros casos van a representar prendas de vestir y que las características que queremos almacenar son la temporada, el género destinado y el estilo que representa..

Memoria plana, búsqueda secuencial (HYPO)

Los casos se almacenan secuencialmente en una lista simple o un fichero, estos ficheros pueden ser planos o binarios.

Para lograr una recuperación eficiente, se indexan los casos de la base. En este método los índices se eligen para representar los aspectos más importantes del caso, y la recuperación involucra la comparación de las características consultadas con cada caso de la base de casos.

Este tipo memoria presenta la ventaja de que añadir nuevos casos resulta muy “barato” (rápido y fácil de implementar). No ocurre así con la recuperación de casos, ya que resulta muy lento cuando el número de casos en la base es alto.

Nuestra base de prendas sería de la siguiente manera:

Figura 24

Ciclo de vida CBR

Un sistema típico CBR está compuesto por cuatro etapas secuenciales, las cuales son realizadas cuando se presenta un problema [9]**,** expuestas brevemente a continuación:

1. *Recuperación*: Dado un problema, en esta etapa se selecciona los casos más similares a éste. Para llevar a cabo esta selección, es necesario tener un algoritmo de recuperación (como por ejemplo el de los *K-vecinos*) y una medida de similitud (como puede ser *distancia euclídea*, *distancia Manhattan*, etc).
2. *Reutilización*: Una vez recuperados los casos símiles, se adaptan al nuevo problema, de forma que exista coherencia entre el contexto en el que fue aplicado y el nuevo. Esta adaptación puede no llevarse a cabo si no se desea, ya que puede haber sistemas en los que no se necesite adaptar la solución, sino llevar a cabo la misma que se tomó bajo ciertas circunstancias.
3. *Revisión*: Tras ello, se procede a la aplicación de la solución propuesta al problema real, determinando su validez.
4. *Retención*: Finalmente y una vez adaptada esta solución, se integra el nuevo conocimiento (esto es, problema + conocimiento) en la base de casos. A este proceso se le denomina aprendizaje.

La figura **[Figura 25]** muestra los pasos del sistema indicados anteriormente.

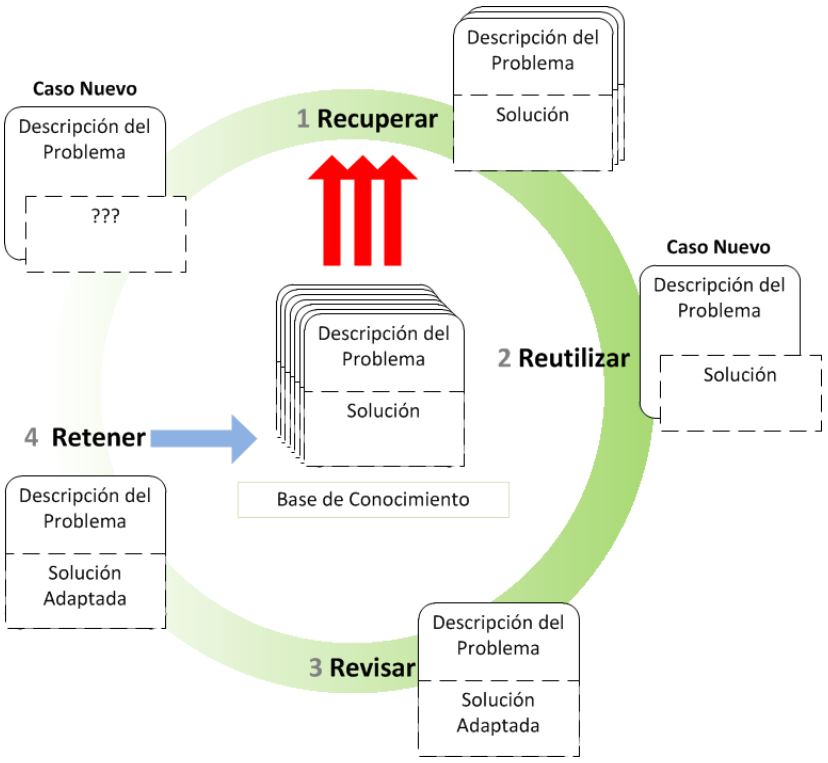


Figura 25

En muchos sistemas CBR no se necesitan almacenar todos los casos existentes, sino que se sigue un criterio para decidir qué casos almacenar y cuáles descartar. A esta operación la denominamos mantenimiento de la base de casos. En determinadas ocasiones, es conveniente mantener una base de casos limitada, ya que así mantendremos ágil el proceso de búsqueda. Por ejemplo, en el desarrollo de un agente inteligente, conviene que la base de casos no sea muy amplia o no se dispare el tamaño, debido a que el tiempo de búsqueda del caso adecuado, puede afectar a la reacción del agente.

Por ejemplo, si disponemos de dos casos muy similares para una misma situación sólo almacenaremos uno de ellos, o podríamos crear un caso artificial que fuese una generalización de dos o más casos concretos.

Ventajas

Algunas de las ventajas que presenta CBR, y que se han considerado de mayor importancia en el contexto del proyecto desarrollado y de bastante interés a nivel general, son las que se exponen a continuación.

Facilitan la organización de la información disponible. Resulta fácil indagar en la base de casos para ver qué experiencias contiene el agente en memoria, en un momento concreto. Ayuda a refinar el comportamiento del bot.

Sistemas dinámicos y adaptativos: el número de casos en memoria incrementa gradualmente, por lo que permite el funcionamiento del sistema en nuevas situaciones. El agente “aprende” nuevas situaciones por adaptación y/o generación de nuevas soluciones.

La posibilidad de poder utilizar casos incompletos o parcialmente definidos facilita la descripción de problemas complejos. Por ejemplo, en un momento dado, los sensores del agente pueden estar no actualizados, digamos por un retraso en la llegada de señales asíncronas.

A medida que el sistema es utilizado, se encuentra más situaciones a las que aplicar nuevas soluciones del dominio. Una vez probadas y analizadas, estas soluciones van siendo incorporadas a la base de casos para así ser utilizadas en futuros problemas.

Consecuentemente, mientras más casos almacenados, se debería poder razonar sobre mayor variedad de problemas del dominio con un nivel más alto de precisión y mayor posibilidad de éxito al aplicar cada una de ellas (menor adaptación).

El número de formas en las que un sistema CBR puede ser implementado es casi ilimitado, por lo que su versatilidad y adaptabilidad a diferentes propósitos es una gran ventaja. No sólo es posible aplicarlo en el ámbito de los videojuegos. De hecho en otros ámbitos es muy recomendable.

Muy simples de implementar. La tarea más difícil es determinar cómo se va a representar un caso y recoger la mayor cantidad de casos posibles como base inicial. En el caso del sistema desarrollado, la representación del caso fue una de las tareas que más tiempo llevó a cabo realizar a la hora de implementar el agente CBR.

Ejemplos

A continuación, se expone algunos ejemplos en los que ha sido aplicada esta técnica así como una breve descripción del papel que desempeña.

American Express - Asignación de riesgos de tarjetas de crédito

American Express, o Amex, es una compañía financiera Americana cuyos principales servicios son los cheques de viajes, seguros, servicios de bolsa, banca en línea y tarjetas de crédito.

Para mejorar el funcionamiento del programa de riesgos en las tarjetas de crédito, se implementó un sistema de asignación de riesgos basado en CBR, en el que se decidía si era o no conveniente conceder crédito a los clientes que así lo solicitasen.

El papel del CBR en este caso, es recoger información sobre créditos concedidos y rechazados anteriormente a personas de perfiles similares al sujeto que se estudia. Se hace una selección del *vecino más cercano*, y se obtiene la decisión que se llevó a cabo en ese momento, y se adapta al problema actual.

La ventaja del uso de CBR en este sistema es que cuantos más casos se analice y se recoja en la base de datos, mejor rendimiento y eficiencia tendrá el sistema global. Así, éste tiene mayor capacidad de acierto para diferenciar entre un buen crédito (con poco o ningún riesgo) de uno malo (cuyo riesgo supera un cierto límite).

Trabajo de Cindy Marling (2009) - Manejo inteligente de la diabetes

El trabajo de Marling *et al.*, tiene por objetivo la creación de un sistema de ayuda a la planificación de terapia basado en CBR que ayude a los pacientes a mantener y mejorar su control reduciendo las tareas de análisis de datos del médico. El sistema CBR utiliza datos sobre el estilo de vida del paciente que pueden influir en las variaciones que se producen en la glucosa en sangre y está enfocado para pacientes en tratamiento con bomba de insulina, con el propósito de proporcionar recomendaciones terapéuticas.

Inicialmente, las recomendaciones se proporcionan al médico para que éste dé su aprobación antes de ser facilitadas al paciente.

Sin embargo, el objetivo final es conseguir un sistema fiable y efectivo, que pueda ser lo suficientemente autónomo para ser integrado en un dispositivo médico del paciente (como por ejemplo, la bomba de insulina) y así facilitar las tareas diarias de control de la diabetes.

El prototipo desarrollado, basándose en datos analíticos en la sangre del paciente, permite encontrar situaciones pasadas en una base de datos, ofreciendo al especialista propuestas de ajustes para la terapia en curso.

Help-desks

Help desks, es una herramienta orientada a la automatización del soporte técnico, cuyo fundamento radica en la aplicación de CBR asociados con algoritmos de análisis semántico y fonético, y la especialización de estructuras para almacenar y recuperar conocimiento de manera eficaz. Ésta da soporte a usuarios de equipos técnicos complejos, proporcionándoles información acerca del uso de dichos equipos y manteniendo sus equipos operativos, llevando a cabo tareas de mantenimiento necesarias y rutinarias.

Los técnicos de Help desks están capacitados para resolver problemas en un plazo corto de tiempo y poseen los conocimientos adecuados en todas las áreas del servicio que presta. A su vez, éstos usan su propia experiencia sobre problemas que han ocurrido en el pasado, para resolver los nuevos inconvenientes que se le presenten al usuario. Sin embargo, a medida de que los sistemas se vuelven cada vez más complejos, las áreas en que los operadores Help-desks tienden a divergir y por lo tanto a ser más costoso sobre todo en tiempo a encontrar una solución al problema dado.

El papel de CBR entra en juego cuando usuarios a los que le urge soporte técnico en un asunto en concreto, no tienen la capacidad de conectar directamente con un especialista en el área, debido a que no esté disponible.

Bot con CBR

CBR significa "Razonamiento Basado en Casos" y es una técnica para encontrar soluciones a los problemas mediante la reutilización de conocimiento en forma de experiencias previamente almacenadas. Es adecuado cuando es imposible, difícil o costoso de definir las reglas exactas.

En principio, es muy simple:

- Definir qué atributos son relevantes para el dominio del problema actual (caso).
- Reunir experiencia - almacenar los valores que componen un caso.
- Cuando surge un nuevo problema, encontrar el "más parecido" de los casos almacenados.

Esta pequeña definición explica con mucha sencillez el Razonamiento Basado en Casos. Y esta sencillez, es el criterio principal a la hora de elegir un motor o librería de CBR para el desarrollo de una inteligencia artificial para un bot en el UT2004.

Muchas librerías permiten implementar un motor de búsqueda basado en casos con pocos requerimientos previos y/o esfuerzo. Algunas además son lo suficientemente ligeras y rápidas para que pueda interactuar en el bucle principal de un juego como el UT2004 sin ningún percance para la movilidad o reacción del bot.

Esta rapidez y ligereza es primordial a la hora de la jugabilidad e interacción, ya que de lo contrario el bot tardaría demasiado a la hora de responder para la realización de algún movimiento. Si ello fuera así se notaría por parte de los demás jugadores como movimientos discontinuos y/o robóticos.

Librería utilizada: FreeCBR

FreeCBR es un motor CBR de código abierto implementado en Java. Se distribuye bajo una licencia de dominio público. Por lo tanto, dispone de total libertad para ser usada y modificada a sus necesidades.

Este software viene en un paquete el cual contiene una aplicación independiente GUI, una aplicación de línea de comandos, una aplicación web, un bean Java, un componente nativo ActiveX MS y un API de desarrollo.

Este motor CBR calcula la coincidencia más cercana entre un conjunto casos. Cada caso consta de un conjunto predefinido de características. Las características de cada caso se definen por un nombre y un tipo de dato, donde el tipo de dato puede ser: String, MultiString, Float, Int y Bool. Por ejemplo “nivelDeVida”, “int”.

La coincidencia más cercana se calcula utilizando la ponderación de la distancia euclidiana.

Como el teorema de Pitágoras en n dimensiones.

El motor devuelve este valor como un "porcentaje de coincidencia" que se calcula como “**100 * (1 - sqrt (distancia del caso / sum (pesos)))**“ y recibe un valor entre 0 y 100.

La distancia entre la búsqueda y un caso es un número de coma flotante entre 0 y 1 y se calcula como: “**distancia del caso = peso1 * dist12 + peso2 * dist22 + ... + pesoN * distn2**”.

Donde:

- **distX:** Es la distancia entre la característica buscada y la característica del caso actual. Este valor es un flotante entre 0 y 1, donde 0 significa resultados exactos y 1 la distancia máxima.
- **pesoN:** Es el peso para la característica "i". Es un número entero mayor o igual a 0, por defecto es 5.

Esto significa que la distancia total de casos es mayor o igual a 0 (0 significa coincidencia exacta) y menor o igual a la raíz cuadrada de la suma de los pesos de las características buscadas.

La distancia entre la característica buscada y la característica del caso actual es calculada como:

- Si el valor del caso o el valor buscado es “?” la característica del caso es descalificada y no se incluye en el resultado.
- Si la búsqueda es sólo “?” no hay coincidencias.

El algoritmo “normal” (llamado el “algoritmo de la distancia normal”) es: “**distancia = min(1, diff(valor buscado, valor del caso)/((valor máximo - valor mínimo) * constante infinita))**”.

En otras palabras, si la diferencia del valor buscado con el valor del caso es mayor a la resta del valor máximo y mínimo multiplicado por la constante infinita entonces la distancia es 1, sino la distancia es: "**diff(valor buscado, valor caso)/((valor máximo - valor mínimo) * constante infinita**".

Donde la "contacte infinita" es una constante que define qué distancia es considerada como infinita.

El algoritmo logarítmico (llamado el algoritmo de la "distancia logarítmica") es: "**ln(distancia normal * (e-1) + 1)**".

La búsqueda se puede realizar para:

- "=": Si es una coincidencia exacta (los valores son iguales) entonces la distancia es 0 sino se usan distintos algoritmos como el algoritmo de "distancia normal" o algoritmo de "distancia logarítmica" o incluso casos estrictos como si no es exacto se descarta.

- "!=": Si es una coincidencia exacta (los valores son distintos) entonces la distancia es 0 sino se usan distintos algoritmos como el algoritmo de "distancia normal" o algoritmo de "distancia logarítmica" o incluso casos estrictos como si no es exacto se descarta.

- ">=": Si es una coincidencia exacta (los valores son mayores o iguales) entonces la distancia es 0 sino se usan distintos algoritmos como el algoritmo de "distancia normal" o algoritmo de "distancia logarítmica" o incluso casos estrictos como si no es exacto se descarta.

- ">": Si es una coincidencia exacta (los valores son mayores) entonces la distancia es 0 sino se usan distintos algoritmos como el algoritmo de "distancia normal" o algoritmo de "distancia logarítmica" o incluso casos estrictos como si no es exacto se descarta.

- "<=": Si es una coincidencia exacta (los valores son menores o iguales) entonces la distancia es 0 sino se usan distintos algoritmos como el algoritmo de "distancia normal" o algoritmo de "distancia logarítmica" o incluso casos estrictos como si no es exacto se descarta.

- "<": Si es una coincidencia exacta (los valores son menores) entonces la distancia es 0 sino se usan distintos algoritmos como el algoritmo de "distancia normal" o algoritmo de "distancia logarítmica" o incluso casos estrictos como si no es exacto se descarta.

- "max": Si es una coincidencia exacta (el valor máximo) entonces la distancia es 0 sino se usan distintos algoritmos como el algoritmo de “distancia normal” o algoritmo de “distancia logarítmica” o incluso casos estrictos como si no es exacto se descarta.

- "min": Si es una coincidencia exacta (el valor mínimo) entonces la distancia es 0 sino se usan distintos algoritmos como el algoritmo de “distancia normal” o algoritmo de “distancia logarítmica” o incluso casos estrictos como si no es exacto se descarta.

Integración de la librería FreeCBR

FreeCBR ya provee un API lo suficientemente simple y potente para realizar la integración con nuestro bot, sin la necesidad imperativa de desarrollar algún tipo de puente o interfaz encargada de conectarse con el motor de CBR de la librería.

Sin embargo para mayor comodidad, simplicidad y control de las acciones a realizar en la base de casos, hemos creado una clase que se encargue de las siguientes tareas: como la de cargar la base de casos, insertar casos nuevos, modificarlos, etc.

Esta clase “ExtFreeCBR” al instanciarla carga automáticamente todos los casos de la base de conocimiento.

```
constructor {
    var motorCBR = nuevaInstanciaFreeCBR();
    motorCBR.cargarCasos(nombre_base_de_casos);
}
```

Luego en cada iteración llamamos a un método “logic” que automáticamente hace las cuatro fases del ciclo CBR (recuperación, evaluación, adaptación y aprendizaje).

```
var resultados = recuperar(consulta);
var solución = evaluar(resultados);
solución = adaptar(solución, resultados, consulta);
retener(consulta, solución);

devolver solución;
```

Estos métodos son los encargados de llamar al motor CBR para realizar las consultas pertinentes, analizar los resultados, adaptarlos a la situación actual de la partida en curso y retener el caso si resulta ser un caso nuevo. Finalmente la solución se devuelve para que el bot siga el curso de la partida.

A la hora de retener los casos habría que hacer un examen previo para mantener el número de casos de la base limitado y así conservar un tiempo de ejecución máximo, pero siempre razonable para que la inteligencia de nuestro agente sea fluida.

También hemos separado en otra clase lo que llamamos la consulta que será la encargada de generar la estructura de un caso de consulta, sus términos, pesos, escalas y todo lo necesario para que el motor pueda hacer la búsqueda y evaluación en la base de casos.

Representación del caso

Para comenzar a modelar el comportamiento del agente inteligente mediante el razonamiento basado en casos, empezamos transformando nuestro árbol de comportamiento, en una base de casos. Para ello primeramente nos definimos una serie de atributos y una solución.

El caso quedó entonces definido por los siguientes atributos:

- Problema
    - boolean pocaVida
    - boolean enemigoVisible
    - String armaActual
    - boolean tengoPocaMunicion
    - int numeroDeUsos
- Solución
    - String accionRealizada

Si bien tanto armaActual como tengoPocaMunicion no eran utilizadas, el resto nos permitía modelar casi todos los comportamientos del árbol.

Después de esto, ampliamos nuestra base de casos con una nueva serie de atributos, para modelar un comportamiento más fiable en nuestro agente inteligente. Dichos atributos son:

- Problema
    - boolean tengoPocaVida
    - boolean tengoPocaMunicion
    - boolean noTengoArmas
    - boolean hayEnemigoVisible
    - boolean tengoEscudoCompleto
    - boolean tengoAdrenalinaCompleta

    - boolean tengoVidaCompleta
    - boolean tengoMunicionCompleta
    - boolean hayItemsVida
    - int numeroDeUsos
- Solución
    - String accionRealizada

En esta representación, el funcionamiento no varía demasiado de la primera versión de nuestro CBR, si bien los casos contemplados son superiores a la versión anterior.

Recuperación y evaluación de los casos

Para escoger la opción que ejecutará nuestro agente, el CBR hace la búsqueda de los mejores casos de entre todos los que tenemos definidos en nuestra base de casos, devolviéndonos el hit o índice de similitud con los datos que se obtienen del entorno. Tras esto, para elegir el caso más óptimo, nos creamos una función que evalúa los resultados almacenados según su índice de similitud en 3 tipos de resultado distintos: fiables, dudosos e inseguros.

Para los resultados fiables, elegiremos de entre la lista de resultados aquel que nos devuelva un porcentaje de similitud mayor.

En los resultados dudosos de la lista que tenemos utilizaremos aquel resultado que haya sido utilizado con mayor frecuencia por el agente inteligente.

En los resultados inseguros, eliminamos de todas las acciones posibles aquellas que nos han devuelto una similitud menor con respecto al caso con el que estamos tratando. De entre las acciones que nos queden elegimos una acción al azar, ya que no tenemos ninguna acción que se asemeja al caso que describe el agente inteligente.

Adaptación y aprendizaje

Para la adaptación de los casos en la base del bot recibimos las 3 listas de resultados de casos (fiables, dudosos e inseguros). Si bien para los dos primeros no se realiza ninguna adaptación (ya que el tanto por ciento de semejanza con el caso que nos presenta el bot actualmente es “suficiente”), para los resultados inseguros realizamos una serie de comprobaciones:

En primer lugar comprobamos el nivel de vida de nuestro bot, y si hay ítems de vida en la pantalla del juego, la acción a realizar por el bot será “buscar Vida”.

En caso contrario, comprobamos si hay algún enemigo visible y si el bot tiene munición en cuyo caso llamamos a la acción “atacar”.

En caso de que ninguna de las dos condiciones funcione, recurriremos a una acción “merodear”, que se dedica a que el bot busque enemigos en la pantalla ya que no tiene ninguna otra acción a realizar (las dos acciones principales en la partida son “atacar” y “buscarVida”, si no se necesita utilizar ninguna de las dos, el bot no tiene nada “importante” que hacer).

Tras esto las nuevas acciones realizadas se guardarán junto con la consulta correspondiente en caso de los resultados inseguros. Para tener una base de datos más detallada también guardaremos los resultados fiables junto con la acción seleccionada.

Una de las características que podemos contemplar, es que podemos ejecutar el bot con una base de casos vacía que se irá rellenando durante la experiencia del juego e interacción con otros agentes.‰

Cabe destacar que muchos de los sistemas CBR no realizan adaptación. No es necesario que un sistema realice dicha adaptación para funcionar correctamente. De hecho en nuestra base de casos, no es realmente necesario realizar una adaptación, ya que el abanico de acciones que puede realizar el bot es limitado.

Comparativa entre CBR y BTs

A continuación se muestra el trabajo de comparación realizados sobre las dos tecnologías utilizadas tras la implementación de cada uno de los agentes inteligentes.

Comparación en tiempos de ejecución

Aquí realizamos una comparación en un combate uno contra uno, entre el CBR bot y el bot con árboles de comportamiento. En este caso el número de casos del sistema CBR es limitado (sólo 15), y su funcionamiento es generalmente más rápido que el del bot con árbol de comportamiento.

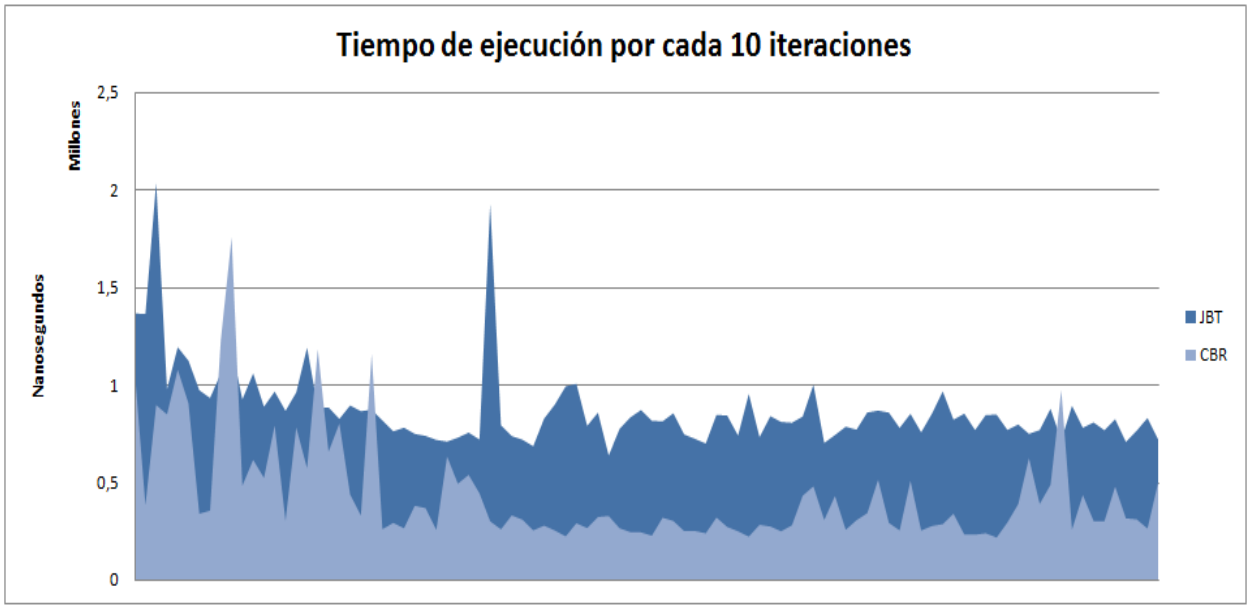


Figura 26

En el siguiente caso comprobamos el funcionamiento del bot con CBR con respecto al bot con árboles de comportamiento, cuando la base de casos es muy grande (alrededor de 2500 casos). Podemos observar que el tiempo de ejecución por cada 10 iteraciones es muy superior en el bot con CBR, ya que al ser tan grande la base de casos, tarda mucho en encontrar el caso idóneo a ejecutar.

A partir de los 10.000 casos guardados en la base de conocimiento, comienza a notarse un retraso en los movimientos y en las acciones del bot. Esto se debe a que cada iteración tarda más de 0.02 segundos en resolverse, más tiempo conforme mayor número de casos haya, y por tanto, retrasa la interacción de la función logic() del bot.

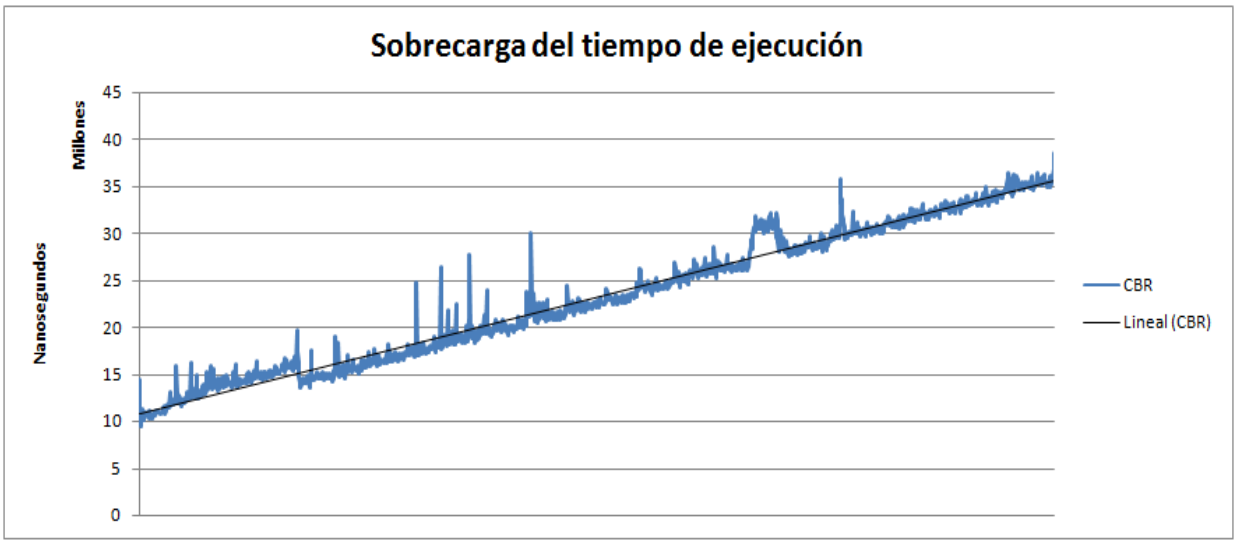


Figura 27

Podemos observar que la sobrecarga en el tiempo de ejecución sigue una línea de tendencia lineal, conforme van aumentando los casos en la base.

Comparativa puntuación

Según nuestra propia experiencia, tras iniciar una partida DeathMatch entre un bot con BTs y un bot con CBR y dejarlos luchando durante un periodo de tiempo indicada en la tabla que se encuentra a continuación, vemos que la puntuación final favorece ampliamente al bot basado en CBR. También indicamos otra serie de detalles como son la estructura en la que se basa la inteligencia del bot, y el tiempo de respuesta máximo a una iteración de la función logic() del bot.

	MUERTES	TIEMPO	TAMAÑO BASE	LATENCIA MAXIMA
CBR BOT	587	2:34:30	46244 casos en la base de conocimiento	87793401 ns
JBT BOT	357	2:34:30	Un árbol de comportamiento	2593762 ns

Figura 28

Otra comparativa realizada es iniciar una partida con 4 bots CBR y 4 bots JBT. Transcurrido un periodo de tiempo de aproximadamente 12 minutos, los resultados obtenidos son:

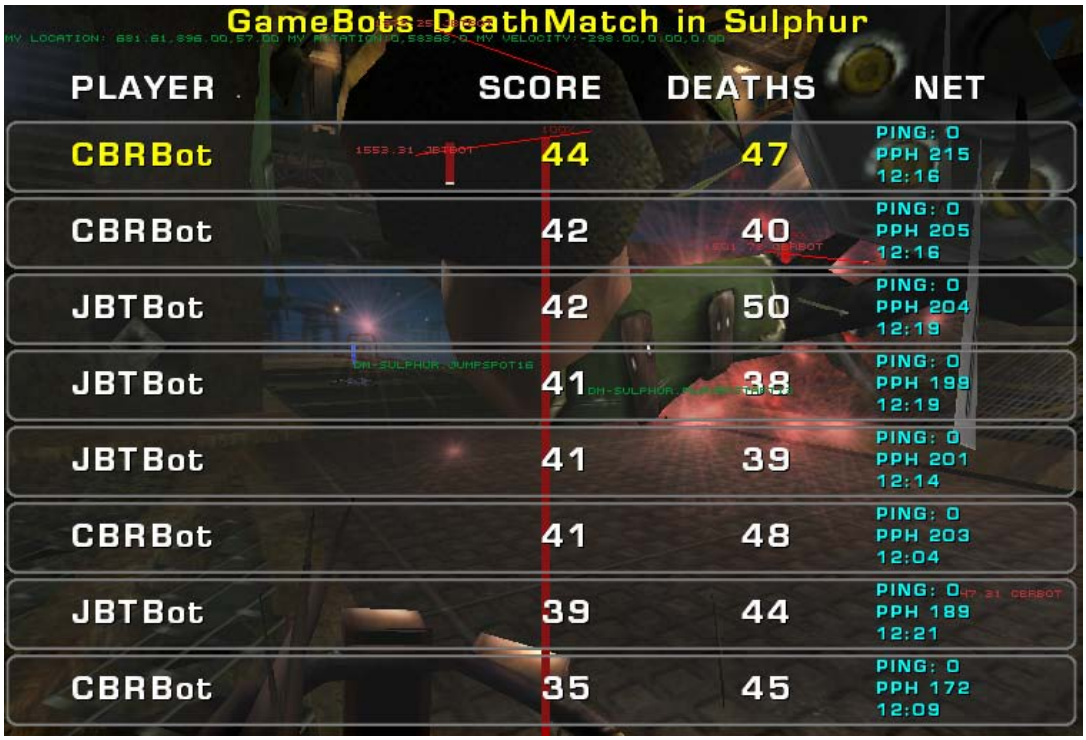


Figura 29

Ventajas y desventajas

En CBR los datos de los casos (features) pueden ser cuantitativos. Permite comparar todas las condiciones que quieres en un momento. Permite detallar mucho más con menos descripción.

BT compara las cosas por tramos o niveles del árbol, es decir, hace el análisis paso por paso (comprueba cada condición una por una). Es más visual de cara al programador, porque le permite comprender de una manera más sencilla el funcionamiento íntegro del agente.

Desde nuestra experiencia hemos deducido que CBR puede resultar muy eficiente de cara al desarrollo de personajes para juegos de “mundo abierto” (rol, aventuras, acción), ya que a pesar de ser una herramienta muy potente para el desarrollo de agentes inteligentes, en un shoot ’em up tiene funciones limitadas, pero en un juego de las características mencionadas

anteriormente, el cual contiene una infinidad de funciones distintas, se puede explotar de un modo mucho más eficiente las características de CBR.

El aprendizaje es fundamental en CBR. Éste aprende conforme van ocurriendo sucesos en la partida (aprendizaje interactivo), por tanto es una memoria dinámica. Por el contrario el conocimiento de los BTs es el mismo desde el principio, la memoria es estática.

En BT se debe tener ya una conducta a seguir (o conductas si se trata de varios árboles), por lo que debe existir conocimiento previo del dominio para diseñarlos. Estos patrones de comportamiento son definidos bajo el contexto del dominio por una persona experta en él. En general, no se da el caso que pueda cambiar de estructura de forma dinámica como la manera en la que un ser humano va desarrollando sus conductas a medida que pasan los años.

En CBR se puede definir parcialmente un estado (como se ha indicado en el apartado de ventajas de Teoría) según se requiera, por ejemplo para anular importancia a ciertos atributos del caso.

Una ventaja de BT es que permite extrapolar comportamientos utilizados en un entorno concreto a otros entornos en los que el contexto sea totalmente diferente. Por ejemplo, un árbol que defina la manera en la que se debe entrar en una habitación se podría reutilizar en distintos tipos de juego al que se enfocó inicialmente.

Conclusiones

Este trabajo se ha centrado en el desarrollo e investigación de varias técnicas para gestionar el comportamiento de un agente inteligente dentro del videojuego Unreal Tournament 2004.

Si bien Unreal Tournament 2004 es un videojuego relativamente antiguo y en el que muchos desarrolladores han puesto en práctica otras técnicas para gestionar el comportamiento de un agente inteligente, como por ejemplo, aquellos desarrolladores que participan cada año en el concurso 2kBotPrize, las técnicas que hemos utilizado siguen estando vigentes en los videojuegos actuales ya que modelan eficientes formas de comportamientos y conductas de un bot.

Con la realización de este proyecto, hemos aprendido la importancia de conocer buenas técnicas que modelen comportamientos de la vida o del propio universo. Ya que para muchos problemas que surgen a la hora de desarrollar un sistema (ya sea un videojuego, un sistema educativo, un simulador, etc.) o incluso a la hora de afrontar el día a día de una persona, estas técnicas son herramientas muy eficaces para su resolución.

Cada una de las técnicas simplifica el problema de una forma muy reducida y organizada, que da una mayor visión del universo en general al que nos enfrentamos. Esta organización separa en pequeños problemas que el intelecto humano puede asumir, lo cual nos permite obtener la mejor solución.

Si tenemos en cuenta que estas técnicas han sido utilizadas para modelar la inteligencia de una persona para su destreza en un juego, se agradece que su integración sea tan simple y sencilla, ya que así la búsqueda de soluciones se puede centrar en el verdadero problema de toda inteligencia artificial, *¡el comportamiento humano!*

Para entender realmente este comportamiento podemos pensar que la mente humana es un caos de decisiones y pensamientos que pueden cambiar a lo largo del tiempo, pero que muchas de esas decisiones se mantienen. Y son precisamente esas decisiones las que podemos identificar y modelar para asemejar la inteligencia de un agente artificial a la de una persona. Para entender mejor esta parte podemos poner como ejemplo la acción al realizar cuando en una partida nos están atacando y tenemos poca vida. Las decisiones de cualquier persona que

más se repiten son seguir atacando mientras escapamos y las de esconderse. Asimismo, éstas son acciones fácilmente modelables con las técnicas de este documento.

Con respecto a los árboles de comportamiento, se trata de una técnica de inteligencia artificial que es realmente útil de cara a la especificación de problemas, debido a su facilidad para su comprensión, reduciendo de gran manera la complejidad del problema. Constituye una forma sencilla de aprender a expresar problemas de gran tamaño, por la facilidad de transformación de la especificación inicial en una estructura en forma de este tipo de árbol.

Se ha profundizado en conceptos relacionados con el razonamiento basado en casos. Esta potente herramienta de inteligencia artificial, nos ha permitido comprender que se puede hacer que un sistema "piense" y sea capaz de evolucionar en su toma de decisiones, dependiendo de las experiencias obtenidas con el paso del tiempo, al igual que una persona actúa de una manera u otra, según las experiencias vividas.

Trabajo futuro

Han surgido ideas en relación con CBR y BTs como por ejemplo dar funcionalidad el aprovechar la potencia de una técnica en el marco de la otra. Esto se plantea como trabajo futuro, ya que puede ser bastante interesante analizar cómo actúa el bot y si existe alguna mejora en su comportamiento.

Este estudio podría enfocarse al desarrollo de distintos árboles de comportamiento de acciones muy predefinidas que sigan diversas estrategias, como por ejemplo estrategias defensivas, ofensivas, etc. junto con una base de casos que contenga cada uno de estos árboles.

En definitiva, la realización de este proyecto sirve para extrapolar las técnicas utilizadas a un contexto global, en el cual se le pueden encontrar infinidad de utilidades en cualquier ámbito que deseemos.

Así mismo nos ha servido para entender mejor sistemas de gran dificultad fácilmente comprensibles para una máquina, y a su vez, ayudar a la máquina a comprender el comportamiento del ser humano.

Bibliografía

[1] http://pogamut.cuni.cz/main/tiki-index.php

[2] http://tecfalabs.unige.ch/IDStechnos/?page_id=192

[3]http://cuni.academia.edu/RudolfKadlec/Papers/1351517/Pogamut_3_Virtual_Humans_Made_Simple

[4]http://pogamut.cuni.cz/pogamut_files/latest/doc/javadoc/cz/cuni/amis/pogamut/ut2004/bot/impl/UT2004BotModuleController.html

[5]http://pogamut.cuni.cz/pogamut_files/latest/doc/javadoc/cz/cuni/amis/pogamut/ut2004/agent/module/sensor/Game.html

[6] http://www.behaviorengineering.org/publications/dromey/Dromey-Chapter-Final-20051.pdf

[7] By Sankar K. Pal and Simon C. K. Shiu, Foundations of Soft Case-Based Reasoning.CH-1, ISBN 0-471-08635-5 Copyright @ 2004 John Wiley & Sons, Inc.

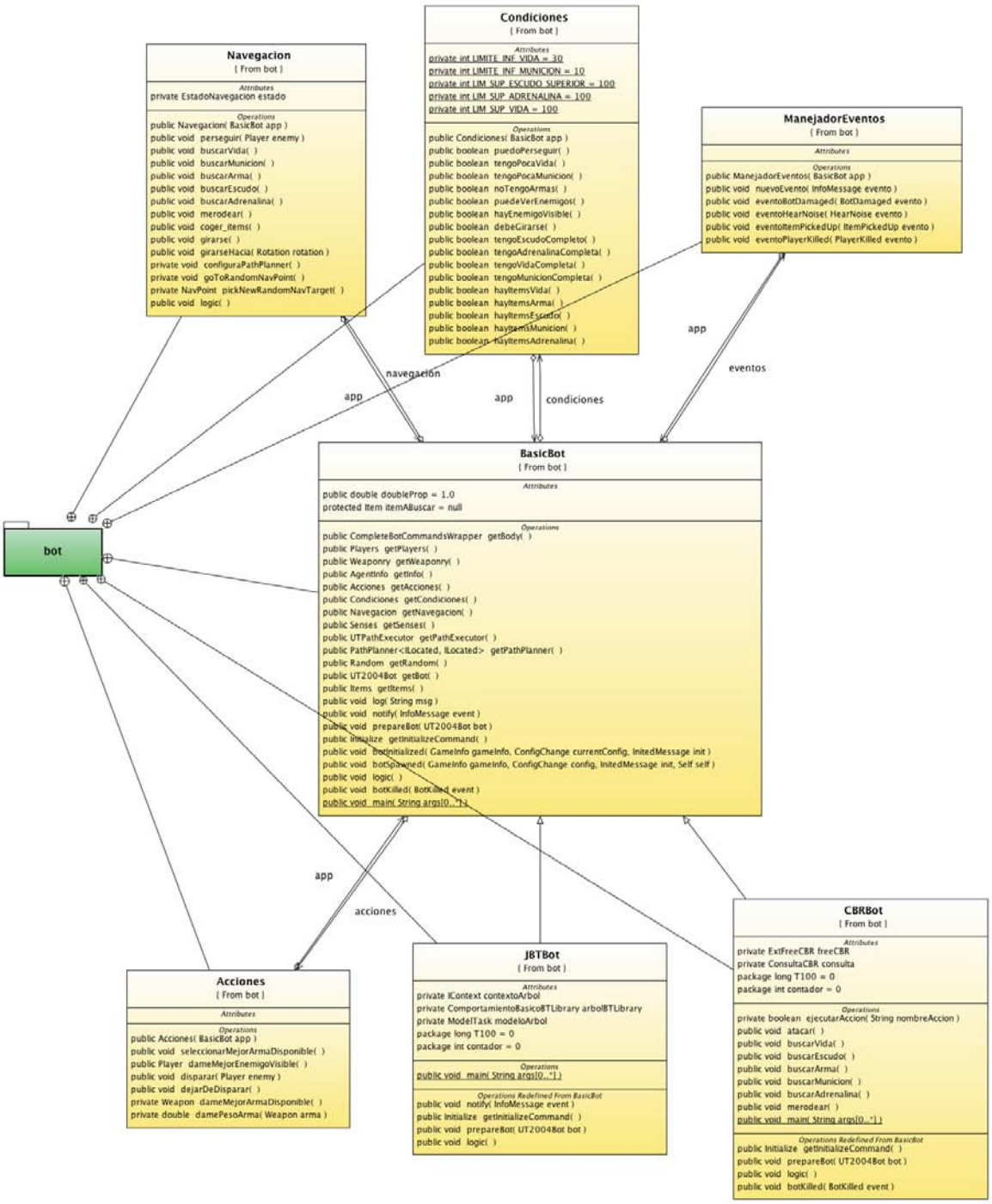
[8] Marling C, Shubrook J, Schwartz F. Toward Case-Based Reasoning For Diabetes Management: A Preliminary Clinical Study And Decision Support System. Prototype. Computational Intelligence 2009; 25(3):165-79.

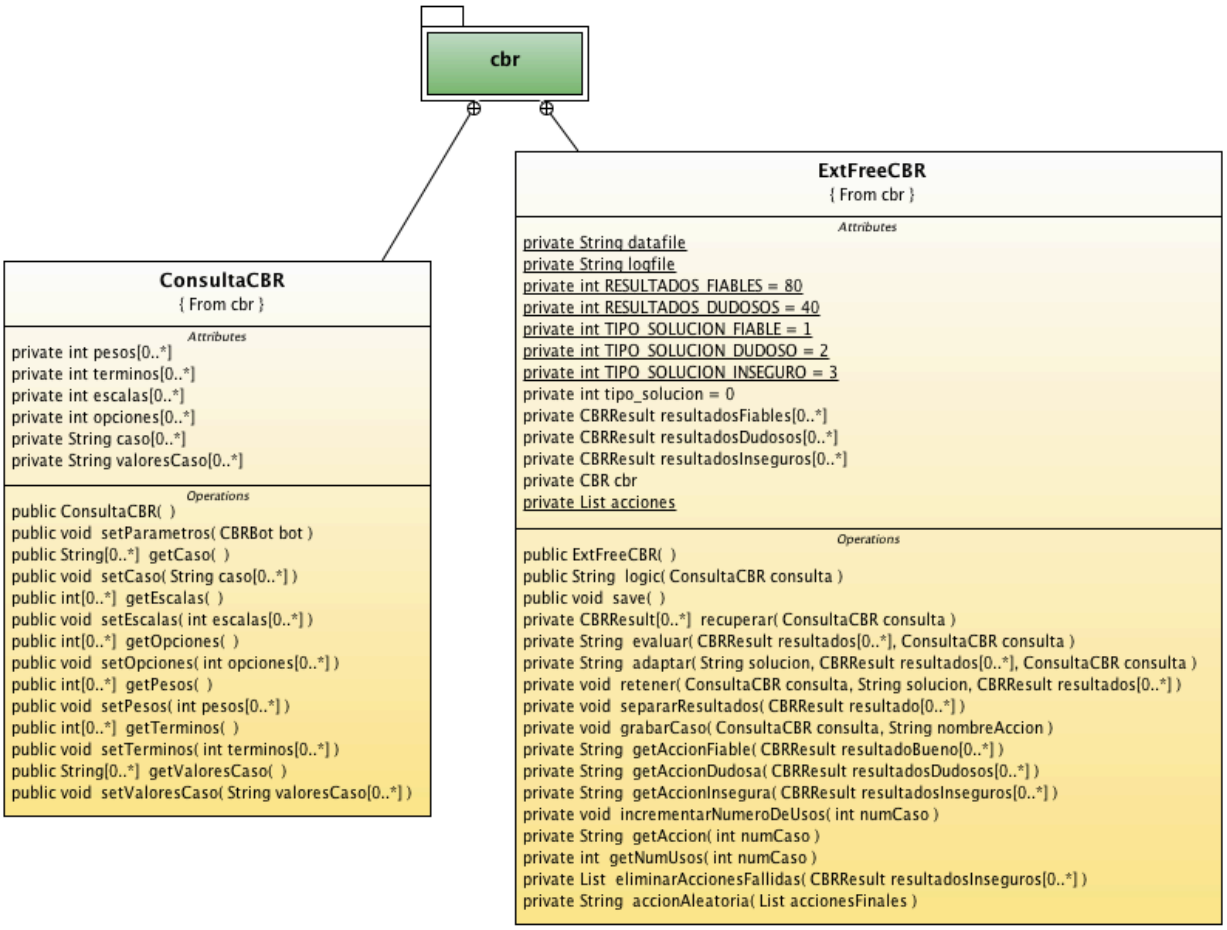
[9] Kolodner, 1993; Aamodt, 1994; Watson et al., 1997.

Anexos

Anexo I - Diagrama UML de clases

A continuación se expone el diagrama con las clases más importantes que hay en el proyecto.




cbr
ExtFreeCBR
{ From cbr }
Attributes
private String datafile
private String logfile
private int RESULTADOS_FIABLES = 80
private int RESULTADOS_DUDOSOS = 40
private int TIPO_SOLUCION_FIABLE = 1
private int TIPO_SOLUCION_DUDOSO = 2
private int TIPO_SOLUCION_INSEGURO = 3
private int tipo_solucion = 0
private CBRResult resultadosFiables[0..*]
private CBRResult resultadosDudosos[0..*]
private CBRResult resultadosInseguros[0..*]
private CBR cbr
private List acciones
Operations
public ExtFreeCBR( )
public String logic(ConsultaCBR consulta)
public void save( )
private CBRResult[0..*] recuperar(ConsultaCBR consulta)
private String evaluar(CBRResult resultados[0..*], ConsultaCBR consulta)
private String adaptar(String solucion, CBRResult resultados[0..*], ConsultaCBR consulta)
private void retener(ConsultaCBR consulta, String solucion, CBRResult resultados[0..*])
private void separarResultados(CBRResult resultado[0..*])
private void grabarCaso(ConsultaCBR consulta, String nombreAccion)
private String getAccionFiable(CBRResult resultadoBueno[0..*])
private String getAccionDudosa(CBRResult resultadosDudosos[0..*])
private String getAccionInsegura(CBRResult resultadosInseguros[0..*])
private void incrementarNumeroDeUsos(int numCaso)
private String getAccion(int numCaso)
private int getNumUsos(int numCaso)
private List eliminarAccionesFallidas(CBRResult resultadosInseguros[0..*])
private String accionAleatoria(List accionesFinales)
ConsultaCBR
{ From cbr }
Attributes
private int pesos[0..*]
private int terminos[0..*]
private int escalas[0..*]
private int opciones[0..*]
private String caso[0..*]
private String valoresCaso[0..*]
Operations
public ConsultaCBR( )
public void setParametros(CBRBot bot)
public String[0..*] getCaso( )
public void setCaso(String caso[0..*])
public int[0..*] getEscalas( )
public void setEscalas(int escalas[0..*])
public int[0..*] getOpciones( )
public void setOpciones(int opciones[0..*])
public int[0..*] getPesos( )
public void setPesos(int pesos[0..*])
public int[0..*] getTerminos( )
public void setTerminos(int terminos[0..*])
public String[0..*] getValoresCaso( )
public void setValoresCaso(String valoresCaso[0..*])

Anexo II - Clases principales del código

Clase Acciones

Esta clase se encarga de definir las acciones que lleva a cabo el bot, como son disparar, escoger la mejor arma, etc.

```
public class Acciones {

    BasicBot app = null;

    public Acciones(BasicBot app) {
        this.app = app;
    }

    public void seleccionarMejorArmaDisponible() {
        app.getWeaponry().changeWeapon((Weapon) dameMejorArmaDisponible());
    }

    public Player dameMejorEnemigoVisible() {
        return app.getPlayers().getNearestVisibleEnemy();
    }

    public void disparar(Player enemy) {
        if (!app.getInfo().isShooting()) {
            this.app.getAct().act(new Shoot().setTarget(enemy.getId()));
        }
    }

    public void dejarDeDisparar() {
        if (app.getInfo().isShooting()) {
            this.app.getAct().act(new StopShooting());
        }
    }

    private Weapon dameMejorArmaDisponible() {
        Weapon mejorArma = null;
        Map<ItemType, Weapon> armasCargadas = null;

        /*
         * Cojo solo las armas que estan cargadas
         */
        armasCargadas = (Map<ItemType, Weapon>)
app.getWeaponry().getLoadedWeapons();
        Iterator<ItemType> iterator = armasCargadas.keySet().iterator();
        while (iterator.hasNext()) {
```

```
            ItemType itemType = iterator.next();
            if (app.getWeaponry().getPrimaryWeaponAmmo(itemType) > 0) {
                if (mejorArma == null) {
                    mejorArma = armasCargadas.get(itemType);
                } else {
                    Weapon candidataMejorArma = armasCargadas.get(itemType);

                    if (damePesoArma(mejorArma) <
damePesoArma(candidataMejorArma)) {
                        mejorArma = candidataMejorArma;
                    }
                }
            }
        }
        return mejorArma;
    }

    private double damePesoArma(Weapon arma) {
        return (double) arma.getDescriptor().getPriDamage();
    }
}
```

Clase Navegación

Esta clase se encarga del movimiento del bot por el mapa así como de la búsqueda de elementos. Todo ello se relaciona en la clase Navegación debido a que implica cambiar la localización del bot.

```
public class Navegacion {

    BasicBot app = null;

    private static enum Estado {

        Nada, BuscarVida, BuscarArma, BuscarAdrenalina, BuscarEscudo,
BuscarMunicion, Merodear, Perseguir
    };

    private class EstadoNavegacion {

        private double MULTIPLICADOR = 20; //segundos
        private double MINIMO = 10; //minimo que dura un estado
        private double duracion; //la duracion del estado
        private double t0; //el timestamp inicial
        private Estado actual; //el estado actual en el que nos encontramos
        private boolean limitado = true; //limitado <=> puedeTerminar

        private EstadoNavegacion(Estado actual, double duracion) {
            t0 = app.getInfo().getTime();
            this.actual = actual;
            this.duracion = duracion;
        }

        public EstadoNavegacion(Estado actual) {
            this(actual, 0);
            double random = MINIMO + app.getRandom().nextDouble() *
MULTIPLICADOR;
            this.duracion = random;

        }

        public EstadoNavegacion(Estado actual, boolean limitado) {
            this(actual, 0);
            this.limitado = limitado;
        }

        public boolean haTerminado() {
            return limitado ? ((t0 + duracion) <= app.getInfo().getTime()) :
false;
        }
```

```
        public Estado getActual() {
            return actual;
        }

        public double getRestante() {
            return t0 + duracion - app.getInfo().getTime();
        }

        // Debug
        public void log() {
            System.out.println("<< " + actual.name() + " ,t0=" + t0 + ",
duracion=" + duracion + (limitado ? ", limitado" : "") + ", ti=" +
app.getInfo().getTime() + " >>");
        }
    }
    private EstadoNavegacion estado;

    public Navegacion(BasicBot app) {
        this.app = app;
        this.estado = new EstadoNavegacion(Estado.Nada, 0);
        configuraPathPlanner();
    }

    public void perseguir(Player enemy) {
        // Si estamos buscando un item y nuestro estado es BuscarVida
        if (estado.getActual() == Estado.Perseguir) {
            return;
        }

        // Si nuestro estado es distinto de BuscarVida, hay que cambiar de
estado inmediatamente
        estado = new EstadoNavegacion(Estado.Perseguir, false);

this.app.getPathExecutor().followPath(this.app.getPathPlanner().computePath(e
nemy.getLocation()));
    }



  public void buscarVida() {
// Si estamos buscando un item y nuestro estado es BuscarVida
        if (app.itemABuscar != null && estado.getActual() ==
Estado.BuscarVida) {
            return;
        }
        // Si nuestro estado es distinto de BuscarVida, hay que cambiar de
estado inmediatamente
        estado = new EstadoNavegacion(Estado.BuscarVida, false);
        List<Item> healths = new LinkedList();
```

```
healths.addAll(this.app.getItems().getSpawnedItems(ItemType.HEALTH_PACK).valu
es());

healths.addAll(this.app.getItems().getSpawnedItems(ItemType.MINI_HEALTH_PACK)
.values());

        app.itemABuscar = DistanceUtils.getNearest(healths,
this.app.getInfo().getLocation());

this.app.getPathExecutor().followPath(this.app.getPathPlanner().computePath(a
pp.itemABuscar));
    }

    public void buscarMunicion() {
        if (app.itemABuscar != null && estado.getActual() ==
Estado.BuscarMunicion) {
            return;
        }
        estado = new EstadoNavegacion(Estado.BuscarMunicion, false);
        List<Item> itemsMunicion = new LinkedList();
        ItemType tipoArmaActual =
this.app.getWeaponry().getCurrentWeapon().getType();

itemsMunicion.addAll(this.app.getItems().getSpawnedItems(tipoArmaActual).valu
es());
        if (!itemsMunicion.isEmpty()) {
            app.itemABuscar = DistanceUtils.getNearest(itemsMunicion,
this.app.getInfo().getLocation());

this.app.getPathExecutor().followPath(this.app.getPathPlanner().computePath(a
pp.itemABuscar));
        }
    }

    public void buscarArma() {

        if (app.itemABuscar != null && estado.getActual() ==
Estado.BuscarArma) {
            System.out.println("<<**** Navegacion buscarArma >>" +
app.itemABuscar.getType().getName());
            return;
        }

        estado = new EstadoNavegacion(Estado.BuscarArma, false);

        System.out.println("<<Navegacion buscarArma>>");
        List<Item> itemsArmas = new LinkedList();
```

```
itemsArmas.addAll(this.app.getItems().getSpawnedItems(ItemType.Category.WEAPO
N).values());
        if (!itemsArmas.isEmpty()) {

            app.itemABuscar = DistanceUtils.getNearest(itemsArmas,
this.app.getInfo().getLocation());

this.app.getPathExecutor().followPath(this.app.getPathPlanner().computePath(a
pp.itemABuscar));
        }
    }

    public void buscarEscudo() {

        if (app.itemABuscar != null) {
            System.out.println("<<**** Navegacion buscarEscudo >>" +
app.itemABuscar.getType().getName());
            return;
        }

        System.out.println("<<Navegacion buscarEscudo>>");
        List<Item> itemsEscudo = new LinkedList();

itemsEscudo.addAll(this.app.getItems().getSpawnedItems(ItemType.Category.ARMO
R).values());
        if (!itemsEscudo.isEmpty()) {

            app.itemABuscar = DistanceUtils.getNearest(itemsEscudo,
this.app.getInfo().getLocation());

this.app.getPathExecutor().followPath(this.app.getPathPlanner().computePath(a
pp.itemABuscar));
            estado = new EstadoNavegacion(Estado.BuscarEscudo);

        }
    }

    public void buscarAdrenalina() {

        if (app.itemABuscar != null) {
            System.out.println("<<**** Navegacion buscarAdrenalina >>" +
app.itemABuscar.getType().getName());
            return;
        }

        List<Item> itemsAdrenalinas = new LinkedList();

itemsAdrenalinas.addAll(this.app.getItems().getSpawnedItems(ItemType.Category
.ADRENALINE).values());
```

```
        if (!itemsAdrenalinas.isEmpty()) {

            app.itemABuscar = DistanceUtils.getNearest(itemsAdrenalinas,
this.app.getInfo().getLocation());

this.app.getPathExecutor().followPath(this.app.getPathPlanner().computePath(a
pp.itemABuscar));
            estado = new EstadoNavegacion(Estado.BuscarAdrenalina);

        }
    }

    public void merodear() {
        System.out.println("<<Navegacion merodear>>");
        if (!this.app.getPathExecutor().isMoving()) {
            estado = new EstadoNavegacion(Estado.Merodear, false);
            goToRandomNavPoint();
        }
    }

    public void coger_items() {
        Item item =
DistanceUtils.getNearest(this.app.getItems().getVisibleItems().values(),
this.app.getInfo().getLocation());
        this.app.getPathExecutor().stop();

this.app.getPathExecutor().followPath(this.app.getPathPlanner().computePath(i
tem));
    }

    public void girarse() {
        this.app.getAct().act(new Rotate().setAmount(32000));
    }

    public void girarseHacia(Rotation rotation) {
        this.app.getAct().act(new Rotate().setAmount((int)
rotation.getYaw()));
    }

    private void configuraPathPlanner() {
        // add stuck detector that watch over the path-following, if it
(heuristicly) finds out that the bot has stuck somewhere,
        // it reporst an appropriate path event
        this.app.getPathExecutor().addStuckDetector(new
StupidStuckDetector(this.app.getWorldView(), 3.0));

        // specify what will happen when the bot reaches its destination or
gets stuck
        this.app.getPathExecutor().addPathListener(new PathExecutorListener()
{
```

```
            public void onEvent(PathEventType eventType) {
                switch (eventType) {
                    case TARGET_REACHED:
                        // most of the time the execution will go this way
                        merodear();

                        break;
                    case BOT_STUCKED:
                        // sometimes the bot gets stucked then we have to
return him
                        // to some safe location, try more complex maps to
get into
                        // this branche
                        NavPoint target = DistanceUtils.getSecondNearest(

app.getWorldView().getAll(NavPoint.class).values(),
                                app.getBot());

app.getPathExecutor().followPath(app.getPathPlanner().computePath(target));
                        break;
                }
            }
        });
    }

    private void goToRandomNavPoint() {
        NavPoint navPoint = pickNewRandomNavTarget();
        //System.out.println("-------------------- <<goToRandomNavPoint>> - "
+ navPoint.toString());
        // find path to the random navpoint, path is computed asynchronously
        // so the handle will hold the result onlt after some time
        PathHandle<ILocated> pathHandle =
this.app.getPathPlanner().computePath(navPoint);
        // make the path executor follow the path, executor listens for the
        // asynchronous result of path planning
        this.app.getPathExecutor().followPath(pathHandle);
    }

    /**
     * Randomly picks some navigation point to head to.
     * @return randomly choosed navpoint
     */
    private NavPoint pickNewRandomNavTarget() {
        //getLog().severe("Picking new target navpoint.");

        // 1. get all known navigation points
        Collection<NavPoint> navPoints =
this.app.getWorldView().getAll(NavPoint.class).values();
```

```
        // 2. compute index of the target nav point
        int navPointIndex = this.app.getRandom().nextInt(navPoints.size());

        // 3. find the corresponding nav point
        int i = 0;
        for (NavPoint nav : navPoints) {
            if (i == navPointIndex) {
                return nav;
            }
            i++;
        }

        // 4. deal with unexpected behavior
        throw new RuntimeException(
                "No navpoint chosen. There are no navpoints in the list of
known navpoints");
    }

    public void logic() {
        estado.log();
        if (estado.haTerminado()) {
            System.out.println("<< ---+++--- Estado Ha terminado ---+++---
>>");
            app.itemABuscar = null;
        } else {
            List<Item> itemsSpawned = new LinkedList();

itemsSpawned.addAll(this.app.getItems().getSpawnedItems().values());
            if (!itemsSpawned.contains(app.itemABuscar)) {
                app.itemABuscar = null;
            }
        }
    }
}
```

Clase Condiciones

Esta clase se encarga de la percepción de las condiciones del estado del juego y del estado del bot.

```
public class Condiciones {

    // Constantes
    private static int LIMITE_INF_VIDA = 30;
    private static int LIMITE_INF_MUNICION = 10;
    private static int LIM_SUP_ESCUDO_SUPERIOR = 100;
    private static int LIM_SUP_ADRENALINA = 100;
    private static int LIM_SUP_VIDA = 100;
    BasicBot app = null;

    public Condiciones(BasicBot app) {
        this.app = app;
    }

    public boolean puedoPerseguir() {
        if (!puedeVerEnemigos()) {
            return false;
        }
        return true;
    }

    public boolean tengoPocaVida() {
        Boolean health = app.getInfo().getHealth() < LIMITE_INF_VIDA;

        if (health == true) {
            return true;
        }

        return false;
    }

    public boolean tengoPocaMunicion() {
        Boolean ammo = app.getInfo().getCurrentAmmo() < LIMITE_INF_MUNICION;

        if (ammo == true) {
            return true;
        }

        return false;
    }

    public boolean noTengoArmas() {
        Boolean weapon = !app.getInfo().hasWeapon();
```

```
        if (weapon == true) {
            return true;
        }

        return !app.getInfo().hasWeapon();
    }

    public boolean puedeVerEnemigos() {
        return (this.app.getPlayers().canSeeEnemies() &&
this.app.getPlayers().canReachEnemies());
    }

    public boolean hayEnemigoVisible() {
        return this.puedeVerEnemigos();
    }

    public boolean debeGirarse() {
        return !(tengoPocaVida() || tengoPocaMunicion());
    }

    public boolean tengoEscudoCompleto() {
        return app.getInfo().getHighArmor() == LIM_SUP_ESCUDO_SUPERIOR;
    }

    public boolean tengoAdrenalinaCompleta() {
        return app.getInfo().getAdrenaline() == LIM_SUP_ADRENALINA;
    }

    public boolean tengoVidaCompleta() {
        return app.getInfo().getHealth() == LIM_SUP_VIDA;
    }

    public boolean tengoMunicionCompleta() {
        return
app.getWeaponry().getCurrentWeapon().getDescriptor().getPriMaxAmount()
                == app.getWeaponry().getCurrentPrimaryAmmo();
    }

    public boolean hayItemsVida() {
        List<Item> items = new LinkedList();

items.addAll(this.app.getItems().getSpawnedItems(ItemType.Category.HEALTH).va
lues());
        return !items.isEmpty();
    }

    public boolean hayItemsArma() {
        List<Item> items = new LinkedList();
```

```
items.addAll(this.app.getItems().getSpawnedItems(ItemType.Category.WEAPON).va
lues());
        return !items.isEmpty();
    }

    public boolean hayItemsEscudo() {
        List<Item> items = new LinkedList();

items.addAll(this.app.getItems().getSpawnedItems(ItemType.SUPER_SHIELD_PACK).
values());
        return !items.isEmpty();
    }

    public boolean hayItemsMunicion() {
        List<Item> items = new LinkedList();
        ItemType tipoMunicionArma =
this.app.getWeaponry().getCurrentWeapon().getDescriptor().getPriAmmoItemType(
);

items.addAll(this.app.getItems().getSpawnedItems(tipoMunicionArma).values());
        return !items.isEmpty();
    }

    public boolean hayItemsAdrenalina() {
        List<Item> items = new LinkedList();

items.addAll(this.app.getItems().getSpawnedItems(ItemType.Category.ADRENALINE
).values());
        return !items.isEmpty();
    }
}
```

Clase Manejador de Eventos

En esta clase controlamos los eventos que suceden durante el transcurso del juego.

```
public class ManejadorEventos {
//
http://pogamut.cuni.cz/pogamut_files/latest/doc/javadoc/index.html?cz/cuni/am
is/pogamut/base/communication/messages/InfoMessage.html

    BasicBot app = null;

    public ManejadorEventos(BasicBot app) {
        this.app = app;
```

```
    }

    public void nuevoEvento(InfoMessage evento) {

        if (evento instanceof ItemPickedUp) {
            // Ha cogido un item
            System.out.println("<<Evento ItemPickedUp>>");
            eventoItemPickedUp((ItemPickedUp) evento);
        }

        if (evento instanceof BotDamaged) {
            // Le han hecho dano
            System.out.println("<<Evento BotDamaged>>");
            eventoBotDamaged((BotDamaged) evento);

        } else if (evento instanceof HearNoise) {
            // Ha escuchado algo
            System.out.println("<<Evento HearNoise>>");
            eventoHearNoise((HearNoise) evento);
        }

        if (evento instanceof PlayerKilled) {
            // Ha muerto un jugador
            System.out.println("<<Evento PlayerKilled>>");
            eventoPlayerKilled((PlayerKilled) evento);
        }

        if (evento instanceof EndMessage) {
            //Ha terminado un mensaje
        }

    }

    public void eventoBotDamaged(BotDamaged evento) {
        if (evento.isBulletHit()) {
            if (evento.getInstigator() != null) {

this.app.getNavegacion().girarseHacia(app.getPlayers().getPlayer(evento.getIn
stigator()).getRotation());
            } else {
                this.app.getNavegacion().girarse();
            }
        }
    }

    public void eventoHearNoise(HearNoise evento) {
        //this.app.getNavegacion().girarseHacia(evento.getRotation());
    }

    public void eventoItemPickedUp(ItemPickedUp evento) {
```

```
        //
        UnrealId i = evento.getId();
        String iS = i.getStringId();
    }

    public void eventoPlayerKilled(PlayerKilled evento) {
        //
    }
}
```

Clase BasicBot

Esta clase es un bot vacio, el cual no realiza ninguna acción.

```
public class BasicBot extends UT2004BotModuleController {
    @JProp
    public double doubleProp = 1.0;
    protected Acciones acciones;
    protected Condiciones condiciones;
    protected Navegacion navegacion;
    protected Item itemABuscar = null;
    protected ManejadorEventos eventos;

    public CompleteBotCommandsWrapper getBody() {
        return body;
    }

    public Players getPlayers() {
        return players;
    }

    public Weaponry getWeaponry() {
        return weaponry;
    }

    public AgentInfo getInfo() {
        return info;
    }

    public Acciones getAcciones() {
        return acciones;
    }

    public Condiciones getCondiciones() {
        return condiciones;
    }

    public Navegacion getNavegacion() {
        return navegacion;
    }

    public Senses getSenses() {
        return this.senses;
    }

    public UTPathExecutor getPathExecutor() {
        return pathExecutor;
    }

    public PathPlanner<ILocated, ILocated> getPathPlanner() {
```

```
        return pathPlanner;
    }

    public Random getRandom() {
        return random;
    }

    public UT2004Bot getBot() {
        return bot;
    }

    public Items getItems() {
        return items;
    }

    public void log(String msg) {
        user.info(msg);
    }

    @EventListener(eventClass = InfoMessage.class)
    public void notify(InfoMessage event) {
        eventos.nuevoEvento(event);
    }

    /**
     * Initialize all necessary variables here, before the bot actually
receives anything
     * from the environment.
     */
    @Override
    public void prepareBot(UT2004Bot bot) {
        acciones = new Acciones(this);
        condiciones = new Condiciones(this);
        navegacion = new Navegacion(this);
        eventos = new ManejadorEventos(this);
    }

    /**
     * Here we can modify initializing command for our bot, e.g., sets its
name or skin.
     * @return instance of {@link Initialize}
     */
    @Override
    public Initialize getInitializeCommand() {
        return new Initialize().setName("BasicBot");
    }

    /**
     * Handshake with GameBots2004 is over - bot has information about the
map in its world view.
```

```
     * Many agent modules are usable since this method is called.
     * @param gameInfo informaton about the game type
     * @param config information about configuration
     * @param init information about configuration
     */
    @Override
    public void botInitialized(GameInfo gameInfo, ConfigChange currentConfig,
InitedMessage init) {
    }

    /**
     * The bot is initilized in the environment - a physical representation
of the bot is present in the game.
     * @param gameInfo informaton about the game type
     * @param config information about configuration
     * @param init information about configuration
     * @param self information about the agent
     */
    @Override
    public void botSpawned(GameInfo gameInfo, ConfigChange config,
InitedMessage init, Self self) {
    }

    /**
     * Main method that controls the bot - makes decisions what to do next.
     * It is called iteratively by Pogamut engine every time a synchronous
batch
     * from the environment is received. This is usually 4 times per second -
it
     * is affected by visionTime variable, that can be adjusted in GameBots
ini file in
     * UT2004/System folder.
     *
     * @throws cz.cuni.amis.pogamut.base.exceptions.PogamutException
     */
    @Override
    public void logic() throws PogamutException {
        navegacion.logic();
    }

    /**
     * Called each time the bot dies. Good for reseting all bot's state
dependent variables.
     *
     * @param event
     */
    @Override
    public void botKilled(BotKilled event) {
        itemABuscar = null;
    }
```

```
    /**
     * This method is called when the bot is started either from IDE or from
command line.
     *
     * @param args
     */
    public static void main(String args[]) throws PogamutException {
        // wrapped logic for bots executiona, suitable to run single bot in
single JVM
        new SingleUT2004BotRunner<UT2004Bot>(BasicBot.class,
"BasicBot").startAgent();
    }
}
```

Clase JBTBot

Esta clase encapsula el funcionamiento del bot con árboles de comportamiento. Se muestra a continuación los métodos más relevantes, que son la inicialización y el método logic.

```
public class JBTBot extends BasicBot {

    private IContext contextoArbol;
    private ComportamientoBasicoBTLibrary arbolBTLibrary;
    private ModelTask modeloArbol;
    long T100 = 0;
    int contador = 0;

    @EventListener(eventClass = InfoMessage.class)
    public void notify(InfoMessage event) {
        eventos.nuevoEvento(event);
    }

    /**
     * Here we can modify initializing command for our bot, e.g., sets its
name or skin.
     * @return instance of {@link Initialize}
     */
    @Override
    public Initialize getInitializeCommand() {
        return new Initialize().setName("JBTBot");
    }

    /**
     * Initialize all necessary variables here, before the bot actually
receives anything
     * from the environment.
     */
    @Override
    public void prepareBot(UT2004Bot bot) {
        super.prepareBot(bot);
        arbolBTLibrary = new ComportamientoBasicoBTLibrary();
        contextoArbol = ContextFactory.createContext(arbolBTLibrary);
        contextoArbol.setVariable("bot", this);
        modeloArbol = arbolBTLibrary.getBT("ComportamientoBasico");
    }

    /**
     * Main method that controls the bot - makes decisions what to do next.
     * It is called iteratively by Pogamut engine every time a synchronous
batch
     * from the environment is received. This is usually 4 times per second -
it
```

```
     * is affected by visionTime variable, that can be adjusted in GameBots
ini file in
     * UT2004/System folder.
     *
     * @throws cz.cuni.amis.pogamut.base.exceptions.PogamutException
     */
    @Override
    public void logic() throws PogamutException {
        if (contador == 10) {
            contador = 0;
            T100 = 0;
        }
        long t1 = System.nanoTime();

        super.logic();
        // Parte de los JBT que se ocupa de ejecutarlo
        IBTExecutor executorAcciones =
BTExecutorFactory.createBTExecutor(modeloArbol, contextoArbol);

        do {
            executorAcciones.tick();
        } while (executorAcciones.getStatus() == Status.RUNNING);

        long t2 = System.nanoTime();
        long t3 = t2 - t1;
        T100 = T100 + t3;
        contador++;

        try {
            String fileName = "times_jbt.log";
            FileWriter fw;
            fw = new FileWriter(fileName, true);
            PrintWriter pw = new PrintWriter(fw);
            if (contador == 10) {
                pw.println(T100 / 10);
            }
            pw.close();
            fw.close();
        } catch (Exception e) {
        }
    }

    public static void main(String args[]) throws PogamutException {
        // wrapped logic for bots executiona, suitable to run single bot in
single JVM
        new SingleUT2004BotRunner<UT2004Bot>(JBTBot.class,
"JBTBot").startAgent();
    }
}
```

Clase CBRBot

Esta clase encapsula el funcionamiento del bot con razonamiento basado en casos. Se muestra a continuación los métodos más relevantes, que son la inicialización y el método logic.

```
public class CBRBot extends BasicBot {

    private ExtFreeCBR freeCBR;
    private ConsultaCBR consulta;
    long T100 = 0;
    int contador = 0;

    /**
     * Here we can modify initializing command for our bot, e.g., sets its
name or skin.
     * @return instance of {@link Initialize}
     */
    @Override
    public Initialize getInitializeCommand() {
        return new Initialize().setName("CBRBot");
    }

    /**
     * Initialize all necessary variables here, before the bot actually
receives anything
     * from the environment.
     */
    @Override
    public void prepareBot(UT2004Bot bot) {
        super.prepareBot(bot);
        freeCBR = new ExtFreeCBR();
        consulta = new ConsultaCBR();
    }

    /**
     * Main method that controls the bot - makes decisions what to do next.
     * It is called iteratively by Pogamut engine every time a synchronous
batch
     * from the environment is received. This is usually 4 times per second -
it
     * is affected by visionTime variable, that can be adjusted in GameBots
ini file in
     * UT2004/System folder.
     *
     * @throws cz.cuni.amis.pogamut.base.exceptions.PogamutException
     */
    @Override
    public void logic() throws PogamutException {
        if (contador == 10) {
```

```
            contador = 0;
            T100 = 0;
        }
        long t1 = System.nanoTime();
        super.logic();

        consulta.setParametros(this);

        String nombreAccion = freeCBR.logic(consulta);
        this.ejecutarAccion(nombreAccion);
        long t2 = System.nanoTime();
        long t3 = t2 - t1;
        T100 = T100 + t3;
        contador++;

        try {
            String fileName = "hola.log";
            FileWriter fw;
            fw = new FileWriter(fileName, true);
            PrintWriter pw = new PrintWriter(fw);
            if (contador == 10) {
                pw.println(T100 / 10);
            }
            pw.close();
            fw.close();
        } catch (Exception e) {
        }
    }

    /**
     * Called each time the bot dies. Good for reseting all bot's state
dependent variables.
     *
     * @param event
     */
    @Override
    public void botKilled(BotKilled event) {
        freeCBR.save();
    }

    private boolean ejecutarAccion(String nombreAccion) {
        java.lang.reflect.Method method;

        try {
            if (condiciones.hayEnemigoVisible()) {
                acciones.disparar(acciones.dameMejorEnemigoVisible());
            } else {
                acciones.dejarDeDisparar();
            }
```

```
            method = this.getClass().getMethod(nombreAccion);
            method.invoke(this);
        } catch (Exception ex) {
            return false;
        }
        return true;
    }

    public void atacar() {
        Player masCercano = info.getNearestVisiblePlayer();
        if (masCercano != null) {
            navegacion.perseguir(masCercano);
        } else {
            navegacion.merodear();
        }
    }

    public void buscarVida() {
        navegacion.buscarVida();
    }

    public void buscarEscudo() {
        navegacion.buscarEscudo();
    }

    public void buscarArma() {
        navegacion.buscarArma();
    }

    public void buscarMunicion() {
        navegacion.buscarMunicion();
    }

    public void buscarAdrenalina() {
        navegacion.buscarAdrenalina();
    }

    public void merodear() {
        navegacion.merodear();
    }

    public static void main(String args[]) throws PogamutException {
        // wrapped logic for bots executiona, suitable to run single bot in
single JVM
        new SingleUT2004BotRunner<UT2004Bot>(CBRBot.class,
"CBRBot").startAgent();
    }
}
```